|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Pattern** | | | | | | | | |
| **Definition** | Ein Pattern funktioniert immer in mehr als einer Situation | | | | | | | |
| Kommunikationsmittel (Effizient kommunizieren) | | | | | | | |
| Beschreibt die Vor- und Nachteile (Engineering Trade-offs) | | | | | | | |
| Beschreibt ein bekanntes Engineering Problem | | | | | | | |
| Beschreibt eine generische Lösung für das Problem | | | | | | | |
| **Variations** | | | | | | | | |
| **Architecture Patterns** |  | | | | | | | |
| **Software pattern** | Design Pattern (GoF), Pattern-oriented software architecture (POSA) | | | | | | | |
| **Other** | Organisational Pattern, Learning patter, documentation pattern, process pattern | | | | | | | |
| **Principles (no patterns)** | | | | | | | | |
| **KIS** | Keep it simple | | | | | | | |
| **DRY** | Don’t repeat yourself | | | | | | | |
| **SOLID** | Single responsibility principle  a class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class).  Open/closed principle  “software entities … should be open for extension, but closed for modification.”  Liskov substitution principle  “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also design by contract.  Interface segregation principle  “many client-specific interfaces are better than one general-purpose interface.”[8]  Dependency inversion principle  one should “depend upon abstractions, [not] concretions.”[8] | | | | | | | |
| **YAGNI** | You Aren’t Gonna Need It | | | | | | | |
| **GoF Pattern** | | | | | | | | |
| **Gang of Four** | Erich Gamma, Richard Helm, Raph Johnson, John Vlissides Design Patterns: Elements of reusable object-oriented Design | | | | | | | |
| **Kritik** | - Patterns sind nicht konkret genug beschrieben - Die Einteilung in die drei Gruppen ist teilweise künstlich - Einige Pattern fehlen  - Zwei Iterator Varianten anstatt unabhängige Pattern (External/Internal)  - Sigleton! | | | | | | | |
| **Creational** | | | | | | | | |
| **Abstract Factory** | Provide an interface for creating families of related or dependent objects without specifying their concrete classes | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.12.15.png | | | | | | | |
| **Prototype** | Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.13.36.png | | | | | | | |
| Ähnlich wie Factory, benutzt aber clone() anstatt new und kann deshalb schneller sein.  Benötigt weniger Klassen wie bei Factory (keine Factory Klasse) | | | | | | | |
| **Singleton** | Ensure a class has only one instance, and provide a global point of access to it.  Wie kann garantiert werden, dass nur ein Objekt von einer Klasse instanziiert und global auf dieses zugegriffen werden kann?  Encapsulated "just-in-time initialization" or "initialization on first use". | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 11.22.30.png | | | | | | | |
|  | Avoids locking overhead: Locks are acquired only when necessary, the most common cases perform fast  /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 13.15.35.png | | | | | | | |
| **Monostate** Singleton Alternative  Idee: nicht die Einschränkung der Anzahl Instanzen welche erstellt werden können, sondern dass sich sämtliche Instanzen identisch verhalten.  Umsetzung: Implement all member variables as static members methods aren't static (but could be)  Vorteile  • Für den Benutzer ist das Monostate-Pattern transparent. Er kann die Objekte normal instanziieren.  • Ableitbar: Es ist möglich die Funktionalität abzuleiten und dadurch zu erweitern.  • klar definierte Richtlinien wann die static-member initialisiert und destruk-  toriert werden.  Nachteile  • Effizienz: Da die Objekte, obwohl deren identischen Verhalten, nicht wiederverwendet werden können, werden viele unnötige Objekte erzeugt.  • Presence: Die static-Member des Monostate Objektes sind immer initialisiert.  • Platform Local: Monostate-Objekte können nicht verteilt werde, da ansonsten die  static-Member Funktionalität verteilt implementiert werden müsste.  • Programer Confusion: Objekte können deren State ändern, ohne dass dies im Anwendungscode ersichtlich ist.  • Aus einer normalen Klasse kann kein Monostate-Klasse erzeugt werden, in dem Vererbung angewendet wird. | | | | | | | |
| **CRTP - curiously recurring template parameter (nicht prüfungsrelevant)** CRTP is when a class inherits from a template class that takes the derived class as template parameter we can apply CRTP to count instances for all your classes that need to limit the number of instances /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 13.24.58.png | | | | | | | |
| **Kritik an Singleton:**  - Lazy Evaluation is a pattern of its own (see POSA 3) | | | | | | | |
| **Factory Method** | Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. FACTORY METHOD is accompanied by DISPOSAL METHOD | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.10.48.png | | | | | | | |
| **PLAIN FACTORY METHOD**  The creator is an object — not necessarily in a class hierarchy — and the **type of the product either is fixed** or varies only with environmental settings or the arguments to the factory method. A PLAIN FACTORY METHOD implementation is normally just a case of providing an ordinary, possibly final or sealed, method that creates instances of another class, with no specific intent to be inherited or overridden. | | | | | | | |
| **CLASS FACTORY METHOD**  The creator is a class rather than an object, and so the factory method is **static**. The creator is often the same class as the product object type, which is not normally defined in a class hierarchy. Direct creation of product objects is often prevented by ensuring that instance constructors are non-public. CLASS FACTORY METHOD pattern is also known as **STATIC FACTORY METHOD** | | | | | | | |
| **POLYMORPHIC FACTORY METHOD**  The possible types of the product object are defined in a class hierarchy. Mirroring the hierarchy of what is created, an interface for creator objects is provided, offering the factory method **abstractly**, and the responsibility for creation is deferred to an implementing subclass. The knowledge of which type of product is required is contracted out to the creator hierarchy, removing the need for a closed and clumsy instanceof solution. This FACTORY METHOD variant is the classic **Gang of Four version**. | | | | | | | |
| **CLONING METHOD**  The product class is the same as the creator class. However, unlike a CLASS FACTORY METHOD the relationship is properly reflexive: the creator is an instance of the class, rather than the class, so that its result is another object of its own type. To be precise, the product is a proper copy of its creator. A CLONING METHOD is a **specific kind of POLYMORPHIC FACTORY METHOD**. | | | | | | | |
| **Builder** | Separate the construction of a complex object from its representation so that the same construction process can create different representations.  Parse a complex representation, create one of several targets. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.26.41.png | | | | | | | |
| Am fügt immer etwas hinzu. Und am Schluss wird das wirkliche Objekt erstellt | | | | | | | |
| **Structural** | | | | | | | | |
| **Adapter** | Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.22.32.png  ttps://upload.wikimedia.org/wikipedia/commons/e/e5/W3sDesign_Adapter_Design_Pattern_UML.jpg | | | | | | | |
| Vereinen von inkompatiblen Interfaces, damit das Basisinterface verwendet werden kann Zwei Formen: Klassen und Objektadapter (Klassenadapter benötigt mehrfach Vererbung → !Java) | | | | | | | |
| **Bridge** | Decouple an abstraction from its implementation so that the two can vary independently.  Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.  Beyond encapsulation, to insulation (= Isolierung) | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.29.16.png | | | | | | | |
| Beliebige Komponenten mit beliebigen Services nutzen Links und Rechts der “Brücke” sind Interfaces (Dependency Injection). Die konkreten Klassen wissen nicht was im Interface steckt | | | | | | | |
| **Composite** | Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.16.20.png | | | | | | | /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.18.19.png |
| **Decorator** | Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.20.30.png | | | | | | | |
| Objekt dynamisch mit zusätzlicher Funktionalität ausstatten Alternative zu Vererbung Unterschied zum Composite: Hier möchte man die Funktionalität des Objekts  ergänzen und nicht ein Baum erstellen | | | | | | | |
| **Flyweight** | Wie kann man das mehrfache Vorkommen von gleichen konstanten Objekten (referenzierten Objekten) vermeiden?  Wald mit vielen Bäumen die gleich aussehen  Ziel mit möglichst wenig Memory, gleiche Objekte darstellen  Flyweight: Ähnliche Objekte mit interner State der immer gleich ist, externer State der sich ändern kann  -> möglichst Memory sparend in Memory speichern | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 11.13.39.png | | | | | | | |
| Extrinsic State = Position, da diese ändert  Intrinsic State = Model, das immer gleich ist (READONLY) 🡪 Concrete Flyweight | | | | | | | |
| **Implementation**:  - Flyweight manager maintains instantiated Flyweights  - Flyweights have to be immutable (read-only)  - Context information is often maintained by parent objec | | | | | | | |
| **Vorteile**:  - reduction of the total number of instances  **Nachteile**:  - can’t rely on object identity | | | | | | | |
| The Flyweight pattern's effectiveness depends heavily on how and where it's used.  Apply the Flyweight pattern when all of the following are true  - An application uses a large number of objects.  - Storage costs are high because of the sheer quantity of objects.  - Most object state can be made extrinsic.  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.  - The application doesn't depend on object identity. (value objects) Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects | | | | | | | |
| **Facade** | Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.24.28.png | | | | | | | |
| Wenn man nicht möchte, dass zu viele externe Klassen auf das Subsystem zugreifen. -> Facade vorschalten Sehr komplexes Subsystem mit einem einheitlichen Interface Nachteil: API’s werden dupliziert | | | | | | | |
| Adapter ist die Schnittstelle vorgegeben  Facade ist immer ein Subsystem, bestehend aus mehreren Komponenten. -> Erleichtert Verwendbarkeit | | | | | | | |
| **Proxy** | Provide a surrogate or placeholder for another object to control access to it.  Use an extra level of indirection to support distributed, controlled, or intelligent access.  Add a wrapper and delegation to protect the real component from undue complexity. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 11.24.29.png | | | | | | | |
| **Behavioral** | | | | | | | | |
| **Chain of Responsibility** | Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.  Launch-and-leave requests with a single processing pipeline that contains many possible handlers.  An object-oriented linked list with recursive traversal. Commonly Used in GUIs to handle mouse clicks and keyboard events | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 11.26.57.png | | | | | | | |
| **Command** | Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations  How can commands be encapsulated, so that they can be parameterized, scheduled, logged and/or undone?  C function pointers are the most primitive Command implementation | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.49.32.png | | | | | | | |
| 1. Kapseln eines Commands im Receiver 2. Invoker führt den Command im Receiver aus | | | | | | | |
| **Vorteile**:  - Ein und derselbe Befehl kann von mehreren Objekten aus aufgerufen werden.  - Neue Befehle sind schnell und einfach erfasst.  - Befehlsobjekte können in einer Befehls-History gespeichert werden (--> Undo/Redo).  **Nachteile**:  - Bei grösseren Designs mit vielen Befehlen können viele kleine Commandklassen entstehen, die das Design zerzausen. | | | | | | | |
| **Interpreter** | Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language  https://upload.wikimedia.org/wikipedia/commons/3/33/W3sDesign_Interpreter_Design_Pattern_UML.jpg | | | | | | | |
| Problem  A grammar for a simple language should be defined  so that sentences in the language can be interpreted.  Solution  Define a grammar for a simple language by defining an Expression class hierarchy and implementing an interpret() operation.  Represent a sentence in the language by an abstract syntax tree (AST) made up of Expression instances.  Interpret a sentence by calling interpret() on the AST. | | | | | | | |
| **Command Processor  (POSA 1)** | Separate the request for a service from its execution. A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.  A separate processor object can handle the responsibility for multiple Command objects  Command Processor complements Command by handling responsibility for execution policy | | | | | | | |
| Sobald mehrere Undo nötig sind Processor = Verwalter des History Stacks Sollte ein eigenen GoF Pattern sein! | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.55.42.png | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.58.46.png | | | | | | | |
| Alle Commands in dem Stack haben eine Reihenfolge 🡪 History  Für ein undo muss pro Do Command zusätzlich der Undo command erstellt werden | | | | | | | |
| **Iterator**  **(External Iterator)** | Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.  The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.  Promote to "full object status" the traversal of a collection.  Polymorphic traversal | | | | | | | |
| Implementierungs Beispiel:  class ObjectIterator {  private Object[] m\_source;  private int m\_current;  public ObjectIterator(Object[] source) {  m\_source = source;  m\_current = 0;  }  public boolean hasNext() {  return m\_current < m\_source.length;  }  public Object next() {  return m\_source[m\_current++];  }  } | | | | Anwendung:  Object[] myList = new Object[] {new Integer(1), new Integer(2), new Integer(3)};  ObjectIterator iterator = new ObjectIterator(myList);  while(iterator.hasNext()) {  System.out.println(iterator.next());  } | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 11.28.11.png | | | | | | | |
| Client ruft next() und hasNext() selbständig auf  **Ablauf:**  1. Initializing an iteration  2. Checking a completion condition  3. Accessing a current target value T  4. Moving to the next target value | | | | | | | |
| **Nachteile:** - robustness hard to achieve when collection changes  - life-cycle management of iterator objects might require a disposal method or iterator must observe its collection  - close coupling between Iterator and corresponding Collection class  - indexing might be more intuitive for "normal" programmers for (no longer?) -> (int i=0; i < c.length(); i++) { ...c[i]...} | | | | | | | |
| **Enumeration Method  (Interner Iterator)** | Support encapsulated iteration over a collection by placing responsibility for iteration in a method **on the collection.** The method takes a Command object that is applied to the elements of the collection | | | | | | | |
| A method on the collection that receives a *Command*, which it then applies to its elements. Command oder besser Lamda Ausdruck  e.g. coll.forEach(o → //do something); | | | | | | | |
| **Implementierungsbeispiel:**  class MyObjectIterator extends ObjectIterator {  public MyObjectIterator(Object[] source) {  super(source);  }  public void print() {  while(hasNext()) {  System.out.println(next());  }  }  public void apply(ObjectHandler handler) {  while(hasNext()) {  handler.handle(next());  }  }  }  interface ObjectHandler {  public void handle(Object o);  } | | | | | | **Anwendung**:  class MyObjectHandler implements ObjectHandler {  public void handle(Object o) {  System.out.println(o);  }  }  Object[] myList = new Object[] {new Integer(1), new Integer(2), new Integer(3)};  MyObjectIterator iterator = new MyObjectIterator(myList);  iterator.print();  iterator.apply(new MyObjectHandler()); | |
| **Vorteile:** - Client is not responsible for loop housekeeping details  - robuster als Iterator, meist zu bevorzugen! | | | | | | | |
| **Batch Method** | Group multiple collection accesses together to reduce the cost of multiple individual accesses in a distributed environment.  Instead of accessing individual elements provide an interface to access groups of elements at once. | | | | | | | |
| **Beispiel**: Beim StringBuilder die toString() Methode | | | | | | | |
| **Mediator** | Define an object that encapsulates how a set of objects interact.  Mediator promotes loose coupling by keeping objects from referring to each other explicitly.  Wie kann die Kommunikation unter Objekten sichergestellt werden, ohne dass die beteiligten Objekte selbst Referenzen auf die anderen Objekte besitzen und so ein Kommunikationschaos vermieden wird?  Wie können komplexe Kommunikationspfade zwischen mehreren Objekten  vereinfacht und somit starke Kopplung verhindert werden?  Unabhängige Instanz, die vermittelt zwischen zwei Parteien. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.40.38.png | | | | | | | |
| **Implementation:**  Mediator as an Observer, Colleagues act as Subject (Observable) | | | | | | | |
| **Vorteile:**  - reusing an object is difficult because it refers and communicates with  many other objects  - a set of objects communicate in well-defined but complex ways  - a behavior that is distributed between several classes should be customizable without a lot of subclassing - Starke Kopplung wird vermieden  - Einzelne Objekte müssen sich nicht kennen.  - Zentrale Stelle für die Steuerung der Kommunikation zwischen Objekten.  - Änderungen an der Kommunikation erfordert keine neue Unterklasse des Objektes.  **Nachteile:** - limits subclassing (of mediator class)  - centralizes control (hard maintainable monoliths, SPoF: Single Point of Failure) | | | | | | | |
| **Memento** | Wie kann ich den internen Zustand eines Objektes erfassen und auslagern, ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurück versetzt werden kann?  Frau verlässt die Familie und hinterlässt Andenken. Care Taker wacht über das Andecken (Mememto)  Sobald sie wieder zurückkommt, kann sie sich an den alten Zustand wieder erinnern | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.44.24.png | | | | | | | |
| **Memento**:  - Stores some or all of the internal state of the Originator.  - Allows only the Originator to access its internal information.  **Originator**:  - Can create Memento objects to store its internal state at strategic points.  - Can restore its own state to what the Memento object dictates. - Der Originator soll seinen State nie public machen  **Caretaker**:  - Stores the Memento objects of the Originators.  - Cannot explore the contents of or operate on the Memento object. | | | | | | | |
| **Vorteile**:  - Der interne Zustand eines Objekts kann gespeichert und jederzeit wiederhergestellt werden.  - Kapselung von Attributen wird beim Erstellen der "Kopie" nicht verletzt.  - Ermöglicht das Extrahieren des internen Status eines Objekts zu einem bestimmten Zeitpunkt.  - Extrahierte Objekte können zu einem späteren Zeitpunkt wiederhergestellt werden.  **Nachteile**:  - Erstellt jedes Mal eine vollständige Kopie der relevanten Daten aus dem Originator. Keine Diffs.  - Auf gespeicherten Zustand kann nicht direkt zugegriffen werden. Der Zustand muss zuerst wiederhergestellt werden.  - Für die Statuskopien wird sehr viel Speicher benötigt, wenn Objekte grosse Attribute besitzen. | | | | | | | |
| **Observer** | Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.04.10.png | | | | | | | |
| **Code State** | primitive Version mit if oder switch statements. U.u performanter als andere Lösungen, da Compiler gut optimieren kann | | | | | | | |
| **Object State / Objects for State** | Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.  Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.  An object-oriented state machine  wrapper + polymorphic wrappee + collaboration  Meist von jeder State Klasse nur eine Instanz → Singleton | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 11.29.40.png | | | | | | | |
| **Method for State** | Each state is a records of method references, which are in the context object  Jeder Block des primitive Switch Statements wird in eine einzelne Methode gepackt → Methoden kommen in eine Tabelle/Map | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 13.19.44.png  macht z.B. sinn, wenn verschiedene States gewisse Funktionalitäten teilen. | | | | | | | |
| **Collection for State** | Partition objects into collections with respect to state, so that state is represented by collection membership | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 13.20.39.png | | | | | | | |
| **Flag for State** | if (a) then stateA, else if (b) stateB) | | | | | | | |
| **Strategy** | Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.05.17.png | | | | | | | |
| **Visitor** | Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es Ihnen, eine neue Operation zu definieren, ohne die Klasse der von ihr bearbeiteten Elemente zu ändern. - Wie kann ich unterschiedliche Operationen, welche auf bestimmten Klassen angewandt werden, realisieren, ohne dass die Klassen bei zusätzlichen neuen Operationen geändert (--> neu kompiliert) werden müssen? - Eine Struktur wird von einem fremden Objekt besucht, ohne zu wissen das es geschieht  - Double dispatch! | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 11.07.37.png | | | | | | | |
| **Kritik**:  - Visitor can be overused: put important stuff into node classes, see Interpreter  - Visitor bad when visited class hierarchy changes: hard to change or adapt existing visitors  - Visitor breaks logic apart (doing similar things to different nodes) | | | | | | | |
| **Nachteile**:  - Starke Kopplung zwischen Visitor und Struktur, weil bei der visit() Methode ein Strukturobjekt übergeben wird | | | | | | | |
| **Template Method** | Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 10.07.25.png | | | /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 13.35.57.png | | | | |
| Implementation:  1. Eine Factory Methode, welche die Struktur vorgibt  2. Mehrere abstrakte Methoden die von den Kindklassen überschrieben werden und von der Factory Methode aufgerufen werden. | | | | | | | |
| **Martin Fowler Pattern** | | | | | | | | |
| **Null Object** | The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior. In short, a design where "nothing will come of nothing”  A Null Object provides a surrogate for another object that shares the same interface but does nothing. Thus, the Null Object encapsulates the implementation decisions of how to do nothing and hides those details from its collaborators | | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 13.24.28.png | | | | | | | |
| **Usage**:  - an object requires a collaborator. The Null Object pattern does not introduce this collaboration--it makes use of a collaboration that already exists  - some collaborator instances should do nothing  - you want to abstract the handling of null away from the client | | | | | | | |
| **Zusammenfassung**  - Introducing a NULL OBJECT simplifies the client's code by eliminating superfluous and repeated conditionals that are not part of a method's core logic.  - The intent of a NULL OBJECT is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior.  **Problem**  - if veschachtelungen, wobei in der Bedingung nur nichts oder default Verhalten implementiert ist.  **Lösung**  - class that conforms to the interface required of the object reference, implementing all of its methods to do nothing or to return suitable default values  - A NULL OBJECT should not be used indiscriminately as a replacement for null references  - Null Object im default Konstruktor setzen  if  \* An object reference may be optionally null and  \* This reference must be checked before every use and  \* The result of a null check is to do nothing or assign a suitable default value  then  \* Provide a class derived from the object reference's type and  \* Implement all its methods to do nothing or provide default results and  \* Use an instance of this class whenever the object reference would have been null  **Konsequenzen**  - Introducing a NULL OBJECT simplifies the client's code by eliminating superfluous and repeated conditionals that are not part of a method's core logic.  - polymorphism and inheritance rather than procedural condition testing  - instances are immutable, and are therefore shareable and intrinsically thread safe.  - NULL OBJECT is typically stateless | | | | | | | |
| **Domain Model** | An object model of the domain that incorporates both behavior and data. - Create a model that defines and scopes a system's business responsibilities and their variations: model elements are abstractions meaningful in the application domain, while their roles and interactions refiect the domain workfiow | | | | | | | |
| **Value Objects** | | | | | | | | |
| **Call by Value / Reference** | | In Java: Default by Reference (Ausnahme primitve Datentypen) In C++: Default by Value, Explizit by Reference mit & | | | | | | |
| **Individual** | | **Event**: Takes place at a particular point in time (MouseClickEvent, Senden einer Bestellung)  **Entity**: The person remains the same, but it can change its properties over time (File, Studenten)  **Value**: Does not change and exists outside time (Buchstaben, Zahlen) | | | | | | |
| **Object categories** | | **Entity**: Entities express system information, typically of a persistent nature. Identity is important in distinguishing entity objects from one another. (ArrayList, Book)  **Service**: Service-based objects represent system activities. Services are distinguished by their behaviour rather than their state content or identity (System.out)  **Value**: For value-based objects interpreted content is the dominant characteristic, followed by behaviour in terms of this state. In contrast to entities, values are transient and do not have significant enduring identity (String)  **Task**: Like service-based objects, task-based objects represent system activities. However, they have an element of identity and state, e.g. command objects and threads. (Task, Runnable, Functional Interface) | | | | | | |
| **Eigenschaften** | | - Gehören zu einem Typ..  - .. innerhalb eines Wertebereichs  - Können ebenfalls eine Dimension besitzen (Meter, KG)  - “pure” OO languages need to mimic values with object classes - Identity is not important - it doesn't matter if a value object is copied or shared by reference - Value classes usually do not have parent classes or children! (final, no virtual)  - value object equality relies on equality of its content 🡪 “a”.equals(“a”); - some value types have natural orderings (numbers, date, string (lexical) - Value übers Netzwerk 🡪 kein Problem | | | | | | |
| **Drei Aspekte** | | 1. identity significant or transparent?  2. object stateful or stateless?  3. does object have significant behavior independent of its state? | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 14.11.16.png | | | | Beispiel: Beim Point ist die Identität sehr wichtig, da er ansonsten ein anderer Punkt repräsentiert! | | |
| **Pattern in Java** | | /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 14.20.03.png | | | | | | |
| **Whole Value** | | Value Objects fehlt es oft an einer Dimension (z.b float oder int)  **Lösung**:  Eigene Typ Klasse, die den Wertebereich und Dimension prüft. Kapseln von nackten Werten in Klassen  -> damit das Typsystem unsinnige Operation prüfen kann | | | | | | |
| **Value Object** | | Klasse als Whole Value = Value Object  **Lösung**: Equals(), hashCode(), toString()(optional), compareTo() (optional) überschreiben und Serializable implementieren Comparisons and ordering between value objects should depend on their content and not on their reference identity. | | | | | | |
| **Enumeration Values**  **→ Java Enum** | | Fixed set of constant values while preserving type safety **Lösung**: - Treat each constant as a Whole Value instance, declaring it as public static final in the Whole Value class  the initialization of each constant should be explicit and fully under the control of the class developer - Prevent public construction  - Exmaple: Month Class | | | | | | |
| **Class Factory Method** | | Simplify construction of Value Objects **Lösung**: Static Method which provides a way of constructing an object (static AObject valueOf(int a) Provide static methods to be used instead of (or as well as) ordinary constructors. | | | | | | |
| **Copied Value** | | How can you pass a modifiable Value Object into and out of methods without permitting callers or called methods to affect the original object? (e.g. date.nextDay())  **Lösung**:  Avoids aliasing problemImplement Clonable Interface or provide copy constructor | | | | | | |
| **Immutable Value** | | How to guarantee no side effect problems **Lösung**: Declare all fields final (even on subclasses) 🡪 No raceconditions possible  - Set the internal state of the Value Class object at construction, and allow no subsequent modifications; i.e., provide only query methods and constructors - no reason to implement Cloneable | | | | | | |
| **Mutable Companion  (Factory Method for immutable values)** | | Simplify complex construction of an immutable value **Lösung**: Implement a companion class that supports modifier methods and acts as a factory for Immutable Value objects (e.g. String and String Builder) | | | | | | |
| **C++** | | Value semantics are in the core of C++ Copy and Move Constructor 🡪 Date(const Date&) | | | | | | |
| **Reflection** | | | | | | | | |
| **Warum?** | | Für mehr Flexibility, Adaptability, Generality 🡪 Exchange software parts | | | | | | |
| **Anwendungen** | | - Testing and Mocking Frameworks exploit Reflection (JUnit, Easymock) - Dynamischer Polymorphismus - Annotations auslesen (OR-Mapper) - Debugging  - Dynamic loading and invocation, RMI, Marshaling  - Exception Handling -> Stacktrace auslesen, Catch-Case (exeption)  - Plugin Systeme -> wo man Komponenten zur Laufzeit hinzufügt  - Dynamically Extend: DLL, Java ClassLoader, OSGI | | | | | | |
| **Nachteile** | | - Huge Performance Impact (Maintain meta information,  - Control flow even harder to understand - More indirection | | | | | | |
| **«Vergleichbares»** | | Eval() in JavaScript, Backticks in Bash | | | | | | |
| **Base- und Meta-Level** | | **Base Level**: Normale KlassenObjekte The base level defines the application logic **Meta Level**: Zusätzliche Meta Informationen The meta level provides a self-representation of the software to give it knowledge of its own  structure and behavior **Meta Object**: Metaobjects encapsulate and represent information about the software.  **MOP Meta Object Protocol** is an interface for manipulating the metaobjects (object.getClass()) | | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 15.30.13.png | | | | | | |
| **Introspection (Read-only)** | | Query object properties | | | | | | |
| **Intercession** | | Modify object properties | | | | | | |
| **Meta Object** | | /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 15.36.15.png | | | | | | |
| **Pattern** (DIY Do it yourself Reflection) | | | | | | | | |
| **Dynamic Polymorphism** | | State, strategy, template method, visitor, null object  (Diese Pattern helfen biem Umsetzen von Reflecion) | | | | | | |
| **Type Object** | | **Problem**:  - You want to categorize objects, eventually dynamically  - Object behavior depends on category, identity must be stable  - Categories should be objects themselves  **Lösung**:  - categorize objects by another object instead of a class - Type can change during run-time by external operation  - Object identity is kept | | | | | | |
| Applied pattern:  /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 15.51.54.png | | | | | | |
| **Vorteile**:  - categories can be added easily, even at run-time  - avoids explosion of (trivial) subclasses  - allows multiple 'meta-levels' (type-objects for type- • objects)  **Nachteile**:  - Confusion because of separation  - Lower efficiency because of indirection | | | | | | |
| **Property List  (Named Arguments**) | | **Problem**:  - Methoden sollen felxibel Parameter entgegen nehmen können (e.g. in java: *method(int …){}* )  - Objekte teilen ihre Attrubute über die Klassen Hierarchie  - Attribute sollen zur Laufzeit hinzugefügt/entfernt werden  **Lösung**:  Property list maps attribute names to values (Key/Value) (vgl Java Map) | | | | | | |
| **Vorteile:**  - Black-box extensibility of attributes by an object's clients  - Attributes can be defined (dynamically) on a per-object basis  - Yet-unknown attributes/extensions can be added later on  - Object extension while keeping object identity  - not a new object is needed for adding an attribute at run-time  - If all attributes are represented in the property list, persistency and attribute iteration can be implemented easily  - no need to rely on language's reflection mechanism  - same "attribute" in PL across hierarchies (not necessarily in common root class  - Flexible parameters for generic method interfaces  **Nachteile:**  - Confusion - different ways to access regular and dynamic attributes  - Type safety left to the programmer (get returns an untyped object)  - Naming not checked by a compiler ('color' vs. 'colour')  - Semantics of attributes not given by class code only by clients  - Null-Object pattern may be hard to apply for attributes because of generic interface  - Run-time overhead can be substantial because of name-lookup  - Memory Management: should get return a reference or a copy of an PL value | | | | | | |
| **Anything** | | | A self-describing data container component  A recursively structured Property List with arbitrary nesting and a simple external format (z.B JSON) | | | | | |
| Anything kann ein Objekt, Liste oder selber ein Anything sein. Anything’s can contain  - simple types (Strings, Numbers, Objects) -> typeid access via AsString(default)  - sequences of Anything’s (arrays)  - associative arrays (dictionaries) | | | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 16.04.04.png | | | | | |
| **Vorteile**:  - Easy and flexible to use, automatically memory managed (C++)  - Readable streaming format, exceptionally good for configuration data  - Universally applicable, flexible interchange across class/object boundaries  - Allows really useful interfaces without overloading  **Nachteile**:  - Less type safety, but might ask for individual type  - Intent of parameter elements not always obvious  - Overhead for value lookup and member access  - Dangling elements in configuration data (is that “/foobaz” thing still used)  - No real object, just data | | | | | |

|  |  |
| --- | --- |
| **Extension Interface (Facet)**  **/Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 16.08.28.png** | The Extension Interface design pattern allows multiple interfaces to be exported by a component, to prevent bloating of interfaces and breaking of client code when developers extend or modify the functionality of the component.  Das Extension Interface Design Pattern zeigt einen Weg auf, Anwendungen strukturiert werden, damit diese modifizierbar und erweiterbar sind |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 16.09.39.png |
| **RootInterface**  Stellt allgemeine Funktionalität bereit, welches jedes Extension Interface bereitstellen muss. Die „getExtension“-Methode muss von jeder Komponente implementiert werden, damit jedes Extension Interface in der Lage ist, ein weiteres Extension Interface anzufordern und damit die Rolle zu wechseln.  **Extension Interface**  gruppiert die semantische Funktionalität und reduziert damit Abhängigkeiten. Es erbt vom RootInterface seine allgemeine Funktionalität. Seine Aufgabe ist, die Rolle einer Kontextgruppe dem Client zur Verfügung zu stellen. Wichtig ist, dass die angebotenen Dienste nach außen gekapselt werden.  **Factory**  erzeugt passende Komponente zum angeforderten Interface. Dies erfolgt mit der „create“-Methode.  **Client**  Diese Klasse implementiert die Anwendungsfunktionalität; darunter versteht man, dass der Client Dienste verwendet, welche durch die Extension Interfaces bereitgestellt werden. Der Zugriff des Clients auf den Komponenten-Service erfolgt immer über das passende Extension Interface, niemals direkt auf die Komponente selbst.  **Component**  Die Klasse implementiert die Funktionalität des dazugehörigen Extension Interface bzw. der Interfaces, da sie mehr als ein Extension Interface implementieren kann. Dabei gibt sie bei Aufruf der „getExtension“-Methode die eindeutige (passende) Interface-Referenz zurück. Sie wird von einer zugehörigen Factory erstellt. |
| **Vorteile**:  - Extensibility: Adding interfaces is easier (like adding properties)  - Separation of concerns  - Can prevent bloated class interfaces, by aggregating extension interfaces  - Different clients can perceive an abstraction differently  - Classes needn't be related to have a common interface (facet) !loser coupling, no inheritance reqired  **Nachteile**:  - Clients become more complex (first obtain facet then use it)  - Subject's original interface does not convey intent  - Run-time overhead |
| **Example of Extension itnerface:**  Eclipse needs a mechanism that allows  • adding a service interface to a component without exposing it in the component's type  - allowing future feature additions/changes without breaking client code  • adding behavior to preexisting types such as IFile  - allow even unforeseen extensions of a component  IAdaptable defines a Method *Object getAdapter(Class a)*  → corresponds to *getExtension* of the Extension Interface pattern |
| **Junit** | |
| **TDD ZOMBIES** | "ZOMBIES" helps creating test cases in TDD:  Z – Zero  O – One  M – Many (or More complex)  B – Boundary Behaviors  I – Interface definition  E – Exercise Exceptional Behavior  S – Simple Scenarios, Simple Solutions |
| **Pattern used in JUnit** | /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 17.18.28.png |
| **Command   used for Test Class** | Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. |
| **Template Method**  **Used for TestCase Class** | Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 17.04.29.png |
| **Collecting Parameter**  **Used for TestResult Class** | To return a collection that is the collaborative result of several methods, add a parameter to all the methods that collects the results |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 17.06.49.png |

|  |  |
| --- | --- |
| **Adapter** | Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces  ermöglicht das ausführen von eigens benannten Testmethoden (i.e. xxxTest(){…}) |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 17.12.39.png |
| **Decorator  Used to extend JUnit** | Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality |
| Adapter: passt die schnittstelle and und hat das verhalten identisch  Decorator: hat die selbe schnittstelle und passt das verhalten an |
| **Reflection Applied** | /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 17.15.49.png |
| **Composite  Group many test cases in test suites** | Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 17.17.54.png |
| **Jutland** | |
| **NanoKernel** | /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 17.31.42.png |
| **FiT: Framework for Integrated Tests** | |
| **Summary** | - write executable application tests in HTML tables  - key idea: documents can serve as tests |
| **Procedure** | * the framework processes a page of HTML * it creates a fixture object for each table in the page * fixture uses the table as input to validation code of your choice (read cell values, communicate with your application, check expected values, and marks cells green/red for siccess/failure) |
| **3 Klassen** | /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 17.32.44.png  Fixture: programmer-defined class. traverses the tree of parses, and TypeAdapter  converts testing values (numerics, dates, etc.) to text and back again. Fixtures talk to the application you are testing and mark individual cells red or green if a check passes or fails  Subclasses of Fixture: Most of the work happens here. some predifened ones: ActionFixture, RowFixture (query application for multiple results and compare them against set of xpected values)  Parse: represents the HTML of a document. The constructor accepts a string and recursively constructs a tree of Parse objects, knit together using the fields parts and more. Each Parse object represents some portion of the document: there’s an individual Parse for each table, row, and cell. |
| **Note** | different than other frameworks:  - nearly all methods are public → "open framework"  - Users don’t “plug into” this framework; they subclass a class and override some default actions. Also, there’s no real cover for the framework designer. Technically, all a user needs to call is the doTables method, but the entire traversal sequence, from doTables down to doCell , is public.FIT will have to live with that traversal sequence forever. There’s no way to change it without breaking client code.  - constructor does a lot of work:  - or a long while, I’ve subscribed to the rule of thumb that constructors should not do the bulk of work in a class. All they should do is put an object into a valid state and leave the real work to the other methods. In general, people don’t expect object creation to be an expensive operation, and they are often surprised when it is. However, the construction code for Parse is undeniably elegant  - One of the deepest lessons in design is that you can gain a great deal if you hold the right things constant. |
| **Conclusion** | The alternative that FIT demonstrates is radical: try to make the framework as flexible and  concise as you can, not by factoring it into dozens of classes but by being very careful  about how each class is factored internally. You make methods public so that when users  want to stray from the normal course, they can, but you also make some hard choices, like  FIT’s choice of HTML as a medium. |
| **Framework** | |
| **Eigenschaften** | - Provides hooks for extensions - Defines the control flow (difference to a library)  - Main method within framework - provides ready-made classes to use - Avoid re-inventing the wheel |
| **Prozeduale Anwendungen** | Das Programm wird sequentiell abgearbeitet, z.B. eine Konso- lenanwendung. Die Anwendung ist jederzeit im Besitz des Kontrollflusses und ruft die Bibliothek nur bei Bedarf auf, welche den Kontrollfluss umgehend wieder an das Anwendungsprogramm übergeben. |
| **Event Loop Anwendungen** | Bei Event-Loop-Programs wird die Anwendung in einem stetigen Event-Loop ausgeführt. Ein Beispiel hierfür ist z.B. die Windows UI API. Die Anwendung läuft in einem stetigen Loop und beim Eintre en eines Ereignis verar- beitet Sie diesen und ruft hierbei ggf. noch einige Bibliotheken auf. Anschliessend wartet die Anwendung auf den nächsten Event. |
| **Framework Anwendungen** | Bei Frameworks übernimmt das Framework den Kontrollfluss und dieses bettet die Anwendung ihn den Kontrollfluss ein. Dadurch ist das Fra- mework jederzeit in Kontrolle des Kontrollflusses und ruft den Anwendungscode nur dann auf, wenn dies notwendig ist. |
| **Microframeworks** | Many design patterns that work together (Strategy, Command Processor, Template Method) |
| **Hollywood-Principle** | Don‘t call us, we call you! |
| **Fragile Base Class Problem** | Wenn eine Base Class zu viel Funktionalität hat 🡪 viele Methoden von den Children wandern in den Parent (andere Klassen könnten die Funktionalität ja auch noch brauchen..) |
| **Herausforderungen** | - Framework users implement application code by subclassing of framework classes or implementing their predefined interfaces by application components. Inheritance is one of the strongest couplings between 2 components - A typical framework re-uses its own components 🡪 leads often to close coupling between the framework parts |
| **Vorteile** | - Less code to write.  - More reliable and robust code.  - More consistent and modular code.  - More focus on areas of expertise, less focus on areas of system compatibility.  - Generic solutions that can be reused for other, related problems.  - Improved maintenance and orderly program evolution.  - Improved integration of related programs. |
| **Lock of Evolution** | There are two reasons for lack of framework evolution and improvement:  - no application uses the framework 🡪 no need to improve or evolve  - one or more applications use the framework 🡪 change may break user cod |
| **Overcome Framework Dilemma** | - Think very very hard upfront - Don’t care to much about framework users 🡪 Provide many useful features to make porting a must - Let framework users participate 🡪 Give users time to migrate (deprecate) - Provide simple, flexible interface, good tests and known pattern. |
| **Context Object** | Maximal drei Methodenparameter 🡪 Ansonsten mehrere Parameter in Kontextobjekt verschieben  Alternativen: Propertylist und Anything |

|  |  |  |
| --- | --- | --- |
| **WAM** | | |
| **Zweck** | Werkzeug, Automat, Material  Applikationen (mit GUIs) für Sachbearbeiter bzw. "Knowledge Worker"  Problem:  - Arbeitsfluss nicht strikt vorgegeben  - Mensch braucht Kontrolle über Abläufe  - unterstützende Software muss flexibel erweiterbar und anpassbar sein, um Flexibilität der Arbeit zu widerspiegeln | |
| **Metapher** | Expertenarbeitsplatz mit  Arbeitsbereich auf dem Materialien (Akten, Agenda, Formulare, ...) mit Werkzeugen (Stift, Rechner, Mappen, ...) bearbeitet werden. Zusätzlich: Automaten für repetitive Abläufe (Kopierer, Shredder, Belegscanner, …) Supporting experts (knowledge worker) in their work by creating a work environment with adequate equipment | |
| **Material**  e.g. Eclipse: source files, jar files, | - (virtuelle) Gegenstände, die schliesslich zum Arbeitsergebnis werden.  - werden mit Werkzeugen und Automaten bearbeitet  - repräsentieren domänen- spezifische Konzepte  - müssen für eine spezifische Aufgabe geeignet sein.  - stellen an ihrer Schnittstelle fachliche Funktionalität zur Verfügung und  kennen außer Fachwerten und anderen Materialien keine anderen Elemente der  WAM-Modellarchitektur.  Material entwerfen:  - Materialien sind konzeptuelle Elemente  - Materialien sind abfragbar, manipulierbar, eingebettet  - Fokus auf die fachliche Domäne: Aufgaben des Menschen definieren die Funktionalität der notwendigen Materialien | |
| **Werkzeug**  e.g. Eclipse: Syntax Highlighting, Refactoring, Klassen Browser, Code Auto-Completion | - User kann mit Werkzeug Materialien bearbeiten oder anschauen  - unterstützt wiederkehrende Aufgaben  - kann für versch. Aufgaben nützlich sein  Ein Software-Werkzeug ermöglicht den interaktiven Umgang mit Arbeitsgegenständen  Werkzeug entwerfen:  - Wiederholende Tätigkeiten in der Domäne prägen Werkzeuge  - Ein Werkzeug vereinfacht die Tätigkeit, aber automatisiert sich nicht vollständig  - Werkzeuge unterstützen kurze zusammenhängende Aufgaben, die jeweils algorithmisch implementiert werden können  - Ein Werkzeug hat Identität, Namen und eine graphische Repräsentation  - Werkzeuge zeigen eine Sicht (View) auf das aktuell bearbeitete Material und gibt unmittelbares Feedback über die Veränderung des Materials.  - Werkzeuge haben ein "Gedächtnis" und können Einstellungen speichern.  Ein Werkzeug sollte den Arbeitsprozess nicht •unnötig vorgeben oder einschränken  Jede Teilfunktionalität eines Werkzeugs sollte jederzeit benutzbar sein  Werkzeuge sollten nicht zu gross werden (Kohäsion/Komplexität) | |
| **Automat**  e.g. Eclipse: build Button zum Kompilieren | - erledigt Routineaufgaben, die vorab festgelegt werden können und immer wieder identisch ablaufen.  - benutzt und bearbeitet Materialien und liefert ein definiertes Resultat.  - Menschlicher Eingriff beschränkt sich auf das Starten eines Automaten oder die Abfrage des Zustands. - Automaten operieren auf Materialien analog zu Werkzeugen - Automaten sollten nicht zu komplexe Aufgaben erledigen 🡪 Aufgabe des Werkzeugs | |
| **Technischer Automat** | Klassischer Zustandsautomat für Geräte | |
| **Sonde (Probe)** | Zustandsabfrage für technischen Automat | |
| **Einstellwerkzeug (Adjusting Tool)** | Spezialwerkzeug für technische Automaten zur Wartung und Steuerung. | |
| **Arbeitsumgebung /  Arbeitsplatz (Workspace)** | Eine Arbeitsumgebung ist der Ort, an dem Werkzeuge, Materialien und andere Objekte bereitliegen und domänenspezifisch arrangiert sind.  Arbeitsumgebung entwerfen:  - Gestaltung eines Softwaresystems mit räumlicher Dimension zum organisieren der Gegenstände & zurgechtlegen, ordnen und ablegen - Erlaube Personalisierung  - biete einen Ort wo Werkzeuge, Materialien und andere benötigte Dinge ihren Platz haben  - Das Raumkonzept aus der Domäne wird als konzeptuelle Einheit im Design auch zur Abtrennung repräsentiert | |
| **Fachliche Behälter (Container)**  e.g. Eclipse: jar, zip, .project-file | - beinhaltet, verwaltet, ordnet und liefert Materialien.  - repräsentieren oft einen Arbeitsablauf und sammeln die dazu gehörenden Objekte. - können auch als Materialien angesehen werden (Composite)  - speichern, sammeln, ordnen Materialien  - Verwalten gespeicherte Materialien und geben Auskunft über diese (z.B. wie viele)  - Container können zusammen mit ihren Materialien an andere Orte verschoben werden - Behälter dürfen wie Materialien nur Materialien und Fachwerte kennen.  - vorhandene Collection Klasse verwenden | |
| **Formular (Form)** | Formulare sind konzeptuell eigenständig, weil sie spezielle Materialien darstellen - können generisch realisiert und genutzt werden (nicht für jedes Material einzeln programmiert)  - Abkürzung für "CRUD" Materialien, ohne weitere Funktionalität (Kombination ist möglich) - Formulare sind spezielle Materialien. Sie enthalten hauptsächlich Felder, die ausgefüllt oder gelesen werden können. Dies passiert mit Hilfe generischer Lese- und Schreiboperationen. Die Felder eines Formulars bestehen aus Fachwerten, um die Konsistenz des Formulars prüfen zu können.  Baue (oder benutze) ein generisches Formularsystem, das mit einem Formulareditor leicht neue Formulare erstellen lässt (Composite) | |
| **Fachwert** | erweitern die Menge der Standard-Datentypen und können von allen Elementen der WAM-Modellarchitektur benutzt werden. Sie folgen der Wertsemantik und können aus anderen Fachwerten zusammengesetzt werden. Außer anderen Fachwerten kennen sie keine Elemente der WAM-Modellarchitektur.  Informationen werden nur durch Fachwerte präsentiert und geliefert.  Fachwerte sind Erweiterungen von primitiven. Sie halten Informationen und sind unveränderlich. | |
| **Pattern** |  | |
| **Zusammenhang Werkzeug und Material** |  | |
| **Aspect (Schnittstellen des Materials)** | Aspect: Koppelung von Werkzeug und Material (n:m)  → Repräsentiere jede unterschiedliche Nutzung eines Materials durch einen separaten Aspekt des Materials  - ermöglichen die Bearbeitung mehrerer Materialien mit dem gleichen Werkzeug  - Material muss Aspekt erfüllen um mit einem bestimmten Werkzeug bearbeitet zu werden  - Werkzeuge werden gegen Aspekt (=Interface) programmiert | |
| **Aspect mit Vererbung / Interfaces** |  | |
| **Aspect mit Adapter oder Decorater** |  | |
| **Abgrenzung Werkzeug und Automat** |  | |
| **Trennung von Funktion und Interaktion** | Interaktion: Was man sieht  Funktion: Under the hood  - MVC  “Separation of concerns” für Werkzeuge - Werkzeuge haben eine Handhabungs- und Visualisierungs-Schnittstelle (Interaktion)  - Werkzeuge haben innere Funktionalität zur Materialbearbeitung (Funktion)  → Zur Werkzeugimplementierung trenne es grundsätzlich konzeptionell in FP und IP | |
| **Werkzeug-komposition** | Zerlege komplexe Werkzeuge in Teile. Das Kombinationswerkzeug bietet einen gemeinsamen Kontext für die eingebetteten Werkzeuge (Divide and Conquer)  Um sicherzustellen, dass die Sub- Werkzeuge unabhängig vom Kontext- Werkzeug bleiben und trotzdem das Kontext- Werkzeug über Zustandsänderungen der Sub- Werkzeuge Rückkopplung erhält, muss das Observer Pattern verwendet werden. | |
| **IAK: Interaktions-komponente** | IAK verarbeitet (System/Windows) Events (z.B Scrolling) - Leitete Applikationsspezifische Ereignisse an FK (Funktionskomponente) IAK kennt und nutzt FK, aber FK kennt IAK so wenig wie möglich | |
| **IP: Interaction Part** | Interaktionsteil | |
| **IAT: Interaktionstypen (Interaction Form)** | Um die IAK möglichst unabhängig vom User Interface zu halten werden IAT verwendet IAK tauscht mit IAT nur Domänendaten aus | |
| **FK: Funktions-komponente** | - implementiert Domänenfunktionalität  - bietet Abfragefunktion für IAK über eigenen internen Zustand und den des bearbeiteten Materials.  - verwaltet Arbeitszustand und Werkzeug-"Gedächtnis" abhängig von Benutzeraktionen und Materialzustand  - greift auf ein bestimmtes Material (über den Aspekt) zu  - Verändert Material  - Verwaltet Arbeitskontext zwischen Materialien und dem Werkzeug (Konsistenzerhaltung) | |
| **FP: Function Part** | Funktionsteil | |
| **PF: Presentation Form** | Presentation Forms (PF) implementieren die spezifische Repräsentation und Interaktion der Interaktionstypen/ interaktion forms für den Benutzer  - PFs kapseln die Widgets eines GUI Tookits  - PFs werden ausserhalb eines Werkzeugs verwaltet, sind aber mit den Interaktionsformen des Werkzeugs gekoppelt | |
| /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 09.10.16.png | |
| **Fachwerte (Domain Value)** | Implementiere Klassen für Fachwerte als Value Object. - Immutable Domain Value Objects (mit Anwendung des Flyweight Design Patterns um Speicher zu sparen)  - Handle-Body Idiom um Speicher bei "grossen" Fachwertobjekten zu sparen (C++)  - Fachwerttyp als Stream (Audio, Video)  - Fachwerttyp per Konfiguration (Type Object) | |
| **Eclipse** | | |
| **Eigenschaften** | | Open platform for application development to make building tools easier and to provide common tools |
| **Architektur** | | /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 10.17.07.png |
| **Extension Mechanism (IAdaptable)** | | /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 09.46.42.png |
| **SWT** | | SWT: eine feine Abstraktionsschicht über das OS GUI Toolkit Swing: Performance sehr schlecht  AWT muss relativ viel in C++ programmieren  SWT: Implementierung für jede Plattform |
| **JFace** | | JFace is a UI toolkit with classes for handling many common UI programming tasks (On top of SWT) |
| **Classpath Hell** | | Classpath hell is an unfortunate consequence of dynamic linking of the kind carried out by Java. 🡪 Zuletzt geladene JAR Version wird genommen. |
| **Plugin** | | Each plug-in gets its own class loader  3 important files: - manifest - plugin.xml (Vorteil XML: Statische Analyse durch Eclipse möglich 🡪 Fehlermeldungen) - Class that implements the extension  Eclipse platform manages plug-ins:  - discover installed plug-ins from various disk locations builds the extension registry  - connects extensions and extension points  - only activates plug-ins when needed (lazy loading, if possible) |
| **OSGi (Open Services Gateway initiative)** | | Specifies a component and service model for Java |
| **OSGi Bundles** | | - OSGI Budles haben explizitie Exports und Import 🡪 Managen von Sichtbarkeiten - Bundles define dependencies and export some of their packages - Components (bundles) can be dynamically installed, started, stopped, and uninstalled |
| **Bundle Manifeset** | | MANIFEST.MF file contains the bundle metadata |
| /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 10.26.38.png |
| **OSGi Services** | | Service connect bundles Service is a POJO and can be registered at a Service Registry at bundle start |
| **Extension Points** | | Extension Point collects contributions to the plug-in offering the Extension Point  Each plug-in contributes to at least one Extension Point |
| /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 10.28.04.png |
| **Pattern in Eclipse** | | |
| **Overview** | | /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 10.34.37.png |
| **Singleton** | | Starke Kopplung, Schwer zu testen! |
| /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 10.35.48.png |
| **Proxy** | | - Provide a handle for the resource (z.B Datei vom Filesystem)  - Handle acts as a proxy for the resource - Small value objects without state information  - Can refer to non-existing resource (file not yet created) - Used for lazy loading (each plugin is a proxy) |
| **Adapter** | | IContentProvider und ILabelProvider sind Adapter für die Domänenobjekte |
| **Observer** | | Change Listener |
| **Strategy** | | Layout Algorithmen (GridLayout, RowLayout) |
| **Command** | | IAction lässt sich später ausführen (Ähnlich wie ein Lambda) |
| **LTK: Refactoring Language Toolkit** | | |
| **Beschreibung** | | a language neutral API for refactorings |
| **Merkmale** | | Sprachunabhängige wurden Teile herausgezogen und in Plugin gebündelt (z.B Diff) |
| **Refactoring Lifecycle (Skeleton)** | | /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 10.45.28.png |
| **Template Method Pattern** | | Grundlegender Skeleton wird vorgegeben und von den Kindklassen die Abstrakten Methoden implementiert. Spezifische Refactorings übernehmen die Kindklassen. |
| **Composite Pattern** | | Mehrere Changes (nach Refactoring) werden als Komposite abgebildet 🡪 Preview der Änderungen |
| **Participants** | | Ein Refactoring kann mehrere Komponenten betreffen - A refactoring participant can participate in the condition checking and change creation of a refactoring processor  - Refactorings that change several source files may have impact on some of the other integrated tools  Beispiel: Beim Verschieben von Variablen müssen die Debug Break Points ebenfalls verschoben werden |
| **Refactoring History und Scriptable Refactorings** | | - Reapplying refactorings on a previous version of a code base  - Composing large and complex refactorings from smaller refactorings |
| **Memento Pattern** | | Refactoring Änderungen erfassen und zu einem späteren Zeitpunkt anwenden |
| **CDT: C/C++ Development Tools** | | |
| **Cevelop** | | Foundation for Cevelop |
| **Visitor Pattern** | | Sammeln aller Namen im AST: Abstract Syntax Tree |
| **Parallel Programming** | | |
| **Terminologie** | | UE Units of Execution: Threads oder Prozesse  PE Processing Elements: Prozessoren oder Cores, UE ≥ PE  Vector Unit Eine Vektor Unit erlaubt das Durchführen von Vektorberechnungen. Beispielsweise kann auf einer Vector-Unit eine Vektoraddition in einem Schritt erfolgen. Ansonsten sind für die Addition eines Vektors |⃗v| Schritte notwendig. |
| **SIMD: Single instruction multiple data  SISD: Single instruction, single data  MIMD: Multiple instruction, multiple data  MISD: Multiple instructions, sindle data** | |  |
| **Decomposing Strategy Pattern:** | | How do you go about decomposing your problem into parts that can be run concurrently?  Task decomposition or/and Data decomposition   * problem is large and significant enough to spend effort parallelizing it * understand key data structures used for the problem * understand what parts are compute intensive, so you can focus on |
| **Task Decomposition** | | Problem in Teilaufgaben aufteilen, welche dann parallel ausgeführt werden können.   * Problem als eine Menge von (nahezu) unabhängigen Tasks betrachten * Aktivitäten identifizieren, welche individuell und relativ unabhängig sind   Beispiele: Web-Server, Worker-Pools oder Processing-Pipelines mit asynchronen Filter.   * functional decomposition   + separate tasks call different functions in program * loop splitting   + distinct independent iterations in an algorithm split to tasks * tasks in data-driven decomposition   + odifferent tasks update different chunks of data |
| **Data Decomposition** | | * most compute-intensive part manipulates a large data structure * similar operations applied to different parts of the data structure independently   e.g. matrix multiplication, weather forecast, image/video/signal manipulation |
| **Dependency Analysis Patterns** | | Um Abhängigkeiten der Tasks zu identifizieren sollten die Tasks zuerst gruppiert, dann sortiert und dann kann das Data-Sharing bestimmt werden. |
| **Group Tasks** | | determine temporal dependency e.g. A need result of B => B → A avoid deadlocks/starvation  shows data-flow dependencies |
| **Order Tasks** | | ordering must satisfy constraints used to group tasks. Choose least restrictive ordering based on constraints => more parallelism possibilities |
| **Data Sharing** | | Nach dem nun Tasks entstanden sind, muss geklärt werden, die diese Tasks Daten untereinander austauschen und teilen   * avoid race conditions * avoid synchronization overhead * avoid communication overhead (e.g. DMA to GPU)   Split data into categories   * read-only * effectively local to one task * read-write (multiple-read-single-write, accumulate) |
| **Algorithm Design Space** | | Der Design-Space hilft einem zu entscheiden, welches Design optimal ist, abhängig von der Reihenfolge die man wählen möchte. |
| determine major organization principle | | |
| **Organized By Ordering** | | |
| **Pipeline Processing** | | Tasks in einer definierten Reihenfolge ausführen. Resultat des vorhergehenden Tasks dient als Input für den folge Task. Jeder einzelne Task wird in einem separaten Thread ausgeführt.  Linear Pipeline    non-linear pipeline |
| /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 11.29.14.png |
| **Event-based**  **Coordination** | | * Koordination erfolgt über Events. Dies ist z.B. ein Anwendungsfall bei HTTP-Server, wo die Anfragen event basiert eintreffen. * Server verwendet Event-Loop und wartet auf Ereignisse. * Tritt Ereignis auf, dispatched der Server das Ereignis an konkreten Handler. * Handler selbst kann wiederum Ereignisse auslösen. |

More (Organized by Task) on the next page!

|  |  |
| --- | --- |
| **Organized by Task** | |
| **Task parallelism** | * Tasks werden so verteilt, dass jede Gruppe von unabhängigen Tasks auf einer eigenen UE ausgeführt wird. * Abhängige Tasks werden auf der selben UE ausgeführt um Abhängigkeiten zu garantieren. Tasks auf verschiedenen UE’s haben keine Abhängigkeiten. * Bei der Verteilung auf die UEs darauf achten, dass die Tasks gleichmässig verteilt werden um eine gute Auslastung über sämtliche UEs zu erreichen. |
| **Divide and Conquer** | jedes Teilproblem auf einer eigenen UE ausführen  Am Ende wird das Resultat vom aufgerufenen Thread gemerged.  Beispiel: 7 Tasks (Kästchen), max 4 concurrent laufende Tasks = max 4 Prozessoren ausgelastet |
| **Recursive Data** | Problem  involves an operation on a recursive data structure (such as a list, tree, or graph) that appears to require sequential processing.  e.g. Finding roots in a forest  Solution  Recasting the problem to transform an inherently sequential traversal of the recursive data structure into one that allows all elements to be operated upon concurrently  This recasting may be difficult to achieve (because it requires looking at the original problem from an unusual perspective) and may lead to a design that is difficult to understand and maintain.  Whether the concurrency exposed by this pattern can be effectively exploited to improve performance depends on how computationally expensive the operation is and on the cost of communication relative to computation on the target parallel computer system. |
| **Geometric Decomposition** | Geometrische Aufteilung der Tätigkeiten in Häppchen, welche dann einzeln gerechnet werden  For arrays and other linear data structures, we can often reduce the problem to potentially concurrent components by decomposing the data structure into contiguous substructures, in a manner analogous to dividing a geometric region into subregions  This decomposition of data into chunks then implies a decomposition of the update operation into tasks, where each task represents the update of one chunk, and the tasks execute concurrently.  If the computations are strictly local, that is, all required information is within the chunk, the concurrency is embarrassingly parallel and the simpler Task Parallelism pattern should be used. In many cases, however, the update requires information from points in other chunks (frequently from what we can call neighboring chunks—chunks containing data that was nearby in the original global data structure).  In these cases, information must be shared between chunks to complete the update. |
| **Map Reduce** | **Map:** Map bildet ein Paar, bestehend aus einem Schlüssel k und einem Wert v, auf eine Liste von neuen Paaren ab, welche die Rolle von Zwischenergebnissen spielen  **Reduce:** Sind alle Map-Aufrufe erfolgt bzw. liegen alle Zwischenergebnisse vor, so wird für jede Zwischenwertliste die Funktion Reduce aufgerufen, welche daraus eine Liste von Ergebniswerten berechnet  map: (key1,value1) -> list(key2,value2)  reduce: (key2, list(value2)) -> list(value) |

|  |  |
| --- | --- |
| **Program Structure** | **SPMD**  Single Program Multiple Data. Auf sämtlichen UEs wird das selbe Programm ausgeführt. Die Berechnungen erfolgen jedoch über andere Daten.  **Master/Worker**  Der Master verteilt die Arbeit auf Worker-Prozesse aus einem Pool und sammelt anschliessend die Resultate ein.  **Loop Parallelism**  Ein serielles Programm mit verschachtelten Loops wird parallelisiert. Hierfür werden einzelne Iterationen des Loops parallel ausgeführt. Erfordert, dass die Loops keine Seiteneffekte haben.  **Fork / Join**  Der Elternprozess spawnt mehrere Subprozesse und wartet anschliessend ggf. auf die Resultate der Prozesse. |
| **Usefulness of algorithm structure patterns** | /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 11.43.40.png |
| **Data Structure** | **Shared Data**  how to deal with shared data- locking, etc  **Shared Queue**  thread-safe queue (typically of work items)  **Distributed Array**  decompose n-dimensional arrays and distribute it to Pes different problem-dependent strategies (e.g. tiles, stripes) important with GPU programming! |
| **Reduction algorithms** | Ein Reduktionsalgorithmus kann z.B. in verteilten Systemen zur Reduktion mehrerer Resultate / Nachrichten in ein einzelnes Resultat / Nachricht benötigt werden.  **serial reduction**  Ein Prozess / ein System erhält die Werte sämtlicher anderer Prozesse und rechnet diese zusammen.  **tree based reduction** *log(n) Schritte*  Die Addition erfolgt in einem Tree, wobei bei den Leaves begonnen wird. Jeweils zwei «benachbarte» Prozesse / Systeme addieren ihr Resultat zusammen. In den folgenden Schritten werden die Teilresultate jeweils addiert, bis sämtliche Addition vorgenommen wurden. Dadurch können mehrere Additionen parallel vorgenommen werden.  **recursive doubling**  Das Recursive Doubling hat Ähnlichkeiten mit der tree based reduction. Die Werte werden Baumartig addiert, wobei jedoch sämtliche Zwischenresultate auf alle Prozesse verteilt werden. Dies ist dann angebracht, wenn sämtliche Prozesse das Resultat benötigen. |
|  | |

|  |  |  |  |
| --- | --- | --- | --- |
| **POSA 1** | | | |
| /Users/Michi/Desktop/Bildschirmfoto 2017-12-29 um 09.23.00.png | | | |
| **Übersicht Struktur** | 1. Architektur Patterns    1. Layers    2. Pipes and Filters    3. Blackboard    4. Broker    5. MVC    6. PAC    7. Microkernel | | 1. Design Patterns    1. Whole-Part    2. Master-Slave    3. Proxy    4. View Handler    5. Forwarder Receiver    6. Client Dispatcher Server |
| **Layers** | It is an architectural pattern to structure an application. It creates layers of different abstraction levels upon parts of the application. A layered application is generally easier to maintain and designed for change. A well-known example of layered architectures is the OSI 7-Layer model for network protocols.  1. Context  You are designing a large system in which high-level operations rely on the lower-level ones. Some parts of the system handle low-level issues such as sensor input, reading bits from a file etc. At the other end, there may be user-visible functionalities.  2. Why should we use this  Layering the architecture of a software design requires the application of various best practices. By grouping components into layers, separation of concerns and abstraction levels have to be discussed and defined. The decoupling and abstraction applied improve maintainability and allow better testing. This pattern also allows to adapt to changes in requirements or implementation preventing developers from having to refactor the whole software.  3. Problem  This pattern prevents developers from producing a monolithic sea of objects with high coupling and chaotic dependency structures. After the Domain Model you have modelled your system, and grouped the items/parts but you do not see directly, which parts are high level and which are low level. In order to exchange the parts if needed, as described before, you need to be able to distinguish between the levels and the complexity of the item tasks. New implementations should also be easily put into the leveling without having a part being on all layers.  4. Solution  Layer the system, begin with the lowest level and put the next higher layers on top of each other until you reach the top. You can additionally implement an interface object that forwards the requests from higher level to a lower level.  5. Structure  Each individual layer shields all lower layers from direct access by higher layers.  https://wiki.ifs.hsr.ch/APF/files/structure.PNG  Responsibilities of Layer N-1: provides services used by Layer N, delegates subtasks to Layer N-2  Note: – Stateless layers are far easier to maintain or test.  – Within a layer, the same layer of abstraction should be used.  6. Dynamics (Typical Scenarios)   1. *Top-down Communication / requests* - A client issues a request to Layer N. Since Layer N cannot process the request on its own, it calls the next Layer N-1 for supporting subtasks. 2. *Bottom-up Communication / notifications* - A device driver detects input at Layer 1. The driver translates the input into an internal format and reports it to the Layer 2, which starts interpreting it. 3. This is when the requests only travel through a subset of Layers. i.e. Starting at Layer N but stops at Layer N-2; 4. This is in the opposite direction of Scenario 3. i.e. Starting at Layer 1 but stops at Layer 3; 5. *Communication protocols* - In the following diagram, Layer N of the left stack issues a request. The request moves down through the layers and reaches Layer 1 which then forwards it to the Layer 1 on the right stack. There the request moves up through the layers of the right stack.   https://wiki.ifs.hsr.ch/APF/files/protocol-stacks.PNG  8. Variants / Extensions  - "Relaxed Layered System": lower layers are visible to higher layers than just its next higher layer. used in UNIX and X Window Systems, as it is more performant  - "Layering through inheritance": lower layers are base classes and a next higher layer inherits from the base class. This way it can access the lower layers services. higher layers can modify the lower layer services according to their needs  9. Known uses  Networking protocols are the best-known example. nearly every application uses this pattern, for example a simple 3-tier application with Presentation, Business Logic and Data Access Layer.  10. Consequences  Pro   * Enables incremental coding and testing * A Layer can be reused in multiple contexts. * Support for standardization (standardized interface like POSIX programming interface) * Dependencies are kept local(in layer), this supports the portability of a system and also the testability since you can test particular layers independently of other components. * Exchangeability of individual layer implementations * Design for change: The introduction of new technologies or implementations only affect a limited and well defined set of layers. * Allows to divide and specify implementation goals among development teams   Contra   * Cascades of changing behavior can happen, when the behavior of a layer changes. * Lower efficiency, as a consequence of the layer pattern, data must be transferred through a number of intermediate layers and must be transformed several times. * Unnecessary work, if services performed by lower layers perform excessive or duplicate work not required by higher layer. * Difficulty of establishing the correct granularity of layers: If an upper layer does not fully exploit this patterns potential for reusability. Too many layers introduce unnecessary complexity and overheads. | | |
| **Pipes and Filters** | This architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.  https://wiki.ifs.hsr.ch/APF/files/pipe-filter.PNG  1. Problem  If you are building a system that needs to process a stream of data, you cannot build it as a single component. This is because of the reason that several developers are working on this system and the requirements of the system may change. Therefore, you want to make sure that the components can be exchanged or enhanced easily even by users. You also want to have small components which can be used in a different context and you want non-adjacent processing steps to not share their information and so on.  2. Solution  The 'pipes and filters' (architectural) pattern divides system tasks in several sequential processing steps which are connected through their data-flow. Each processing step is implemented by a filter component. This filter consumes and delivers data incrementally instead of as a big ball of mud. This enables low latency and parallel processing. The sequence of filters combined by pipes is called a 'processing pipeline'.  3. Structure  **Filters**  Filters are the processing units of the pipeline. It enriches, refines or transforms its input data. A filter activity can be triggered by several events:   * the subsequent pipeline element pulls output data from the filter * the previous pipeline element pushes new input data to the filter * most commonly, the filter is active in a loop, pulling its input from and pushing its output down the pipeline   First two events are passive filters whereas the last is an active filter which starts processing on its own as a separate program or thread.  Remark: All UNIX filters are active by this definition.  **Pipes**  Pipes are connections between filters, between data source and the first filter, and between the last filter and the data sink. Pipes synchronize components that need to be joined with a FIFO buffer.  **Data Source**  A data source represents an input to the system. e.g. a file, a network message, a sensor, etc.  **Data Sink**  A data sink collects the results form the end of a pipeline. There are two different data sink types:   * active: pulls results out of the preceding processing stage * passive: allows the preceding filter to push or write the results into it   4. Dynamics  The control flow can be handled by different approaches, depending on the initiating component:   * Scenario 1: Push Pipeline: A data source pushes data into a chain of filters, each stage actively triggering its successor.  Example: Take and process a foto. * Scenario 2: Pull Pipeline: A data sink requires data Example: Encryption tool needs random data from pseudo random number generator, polling a random data source. * Scenario 3: Mixed Push-Pull Pipeline: Combining active and passive filters.  Example: File read with grep, save search results to file. * Scenario 4: Active Pipeline with buffers.  Example: Recording and compressing a video, writing compressed chapters to disk.   5. When should we apply this pattern: Processing data streams  This pattern works well with multistage data flows and processing components, where no shared state between non-adjacent stages is required. It easily satisfies versatile input and output format requirements and provides highly reusable and interchangeable components. As other streaming patterns, this pattern is suited for multi core processing and parallel computing.  6. Advantages:   * Distributed development: each phase can be specified and developed by separate teams. * Design for change: Enhancements can get applied to stages with less ripple effects to other stages. * Rapid prototyping of new pipelines with existing filters. * Flexibility by filter exchange. * Reuse of filter components. * Efficiency by parallel processing: It is possible to start active filters in parallel in a multiprocessor system or network.   7. Disadvantages / Problems:   * Buffer sizes of pipes and processing speeds of filters can have a critical impact on the system's performance and reliability. * Error handling can't be address globally and is often neglected. * Sometimes there is no way around a global state (e.g.: symbol table of an code generator) * Data transformation overhead:   + Using a single type for all filter input and output data to achieve highest flexibility results in data conversion overheads. * Efficiency gain by parallel processing is often an illusion:   + Because the cost for transferring data can be high compared for the cost of processing the data by itself.   + Some filters consume all their input before producing any output.   + Context switchig between threads or processes is generally expensive   + Synchronization between filters via pipes may stop and start filters often.   8. Note:   * Highly Event driven systems shouldn't get split into sequential stages. * While the stream is processed sequentially without knowledge of it's total length, usually a form of termination symbol has to be defined to mark the end of an input stream. (e.g.: End of File) * If all pipes in a pipeline use the same mechanisms (triggering, input / output formats) interchangeability and reusability increases. * Rule of thumb: Filters should do only one thing and one thing well. * graceful and complex error handling include error correction, resynchronization, additional backup buffering | | |
| **Blackboard** | The Blackboard is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.  1. Context  An immature domain in which no closed, functional approach to a solution is known or feasible.  2. Problem  The Blackboard pattern tackles problems that do not have a feasible deterministic solution for the transformation of raw data into high-level data structures. The solutions to these problems usually span multiple fields of expertise and not trivial strategy exists about the combination of each partial solution. Some problem domains work on highly uncertain or approximate data generating multiple alternative solutions in each step, or mind blowingly large solution search spaces.  3. Additional forces   * Immature domain: The problem domain might evolve with the project due to experiments and various approaches. * Variety of algorithms for partial solutions * Parallelism with disjoint algorithms working on the same partial problem.   4. Solution  https://wiki.ifs.hsr.ch/APF/files/blackboard.png  The blackboard pattern builds upon the idea to have independent programs. Each program solves a different problem. But overall the programs work cooperatively together on the solution. Still the individual programs do not call each other and it is not predetermined when each program starts. Moreover, a centralized program starts each needed program upon the current state. Due to that you can create different approaches for finding the solution.  During the problem solving process the problem gets divided into partial problems. These will then generate a partial solution which gets accepted, rejected or changed, and which then are all together combined at the end. The set of all possible solutions is called the \*solution space\*. Potential solutions of all possible solutions are on the highest level.  5. Structure  **Blackboard**   * The blackboard represents the central data store for all related elements of solution space and control data. * The term 'vocabulary' is used to define the set of all data elements that can appear on the blackboard. * In the solution space, the terms 'hypothesis' or 'blackboard entry' are used to represent possible solutions. * Hypotheses are usually annotated with an abstraction level describing their distance to the original input. The lowest abstraction level is similar to input data representation, while the highest abstraction level would be the output. (e.g. output text of speech recognition). * It is often useful to specify relationships between hypotheses, such as part-of or ''in-support-of'. * Estimated degree of truth of hypotheses or the time interval are also other useful hypothesis attributes. * The blackboard can be viewed as a 3 dimensional problem space. (e.g. X-axis: time line of audio data input, Y-axis: abstraction levels, Z-axis: alternative solutions)   **Knowledge Source**   * Knowledge sources are independent subsystems that solve specific aspects of the problem. They do not communicate directly, rather they only read from and write to the blackboard. The solution is built by integrating the results of multiple knowledge sources. * A Knowledge source can operate on forward or backward reasoning depending on the abstraction levels used. The forward reasoning means a particular solution is transformed to a higher-level abstraction. The backward reasoning is used support a solution by searching at a lower level. * Knowledge source are split into a condition-part and an action-part. The conditional part determines if a knowledge source can make a contribution on the current state of the solution process. The action part produces a result that may cause a change on the blackboard.   **Control**   * The control component runs the monitoring loop supervising all changes on the blackboard. Depending on the design/strategy the control component schedules knowledge source evaluations and activations. * In practice, it is more likely that each reasoning step introduces several new hypotheses. The number of possible steps explodes. Therefore it is equally important to restrict the alternatives rather than finding new ones. * The control component has the responsibility to determine when the system should halt (e.g. the threshold/limits of the system's resources are reached or an acceptable solution is found).   6. Dynamics   * Control is triggering knowledge sources to work on specific hypotheses. * Knowledge Sources are observing the blackboard.   7. Advantages:   * Support for changeability and maintainability: This approach allows for dynamic evolution of the problem domain * Experimentation: Organize Experiments with indeterministic algorithms. * Reusable Knowledge: The Blackboard enabled various knowledge sources to communicate in shared formats. * Fault tolerance and robustness: Only solutions (hypotheses) of high certainty survive long enough to make it to output. Uncertain results get rejected.   8. Disadvantages / Problems:   * No good solution guaranteed: Usually Blackboard systems can solve only a certain percentage of their given tasks correctly. * Difficulty of testing: With many intertwined indeterministic behavious, unit testing is not just difficult but often able to verify the correctness. * Low efficiency: Blackboard systems are notoriously inefficient due to high overhead. * High development effort: Empirically these systems require long team evolutions and design iterations. * No support for parallelism: Concurrent access on the same blackboard poses a major problem for control strategy.   9. Examples / Usages:  Diagrams, tables or English phrases, Vision, image recognition, speech recognition and surveillance, artificial intelligence | | |
| **Model View Controller (MVC)** | The Model-View-Controller (MVC) pattern divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.  1. Context  Interactive application with a flexible human-computer interface.  Note: In POSA-1 the pattern is explained in the context of a desktop application. It has many more applications as we will explain below.  2. Problem  Often in Software Development we evolve requirements and implementation of the problem domain, business logic, data structure, the user experience and the interaction design in a non-sequential and parallel way. While working on business logic, changes to the presentation view or the application simply needs to be running on a new platform.  3. Forces   * The same information is presented differently in different windows (eg. bar or pie chart) * It should be possible to integrate different representations of the same data without changing data storage structures. * Changes to the UI should be easy, and even possible at run-time.   4. Solution  The widely known MVC separates an interactive application into three areas: processing, output and input.  The model represents the core data and functionality. It is independent of specific output representation or input behaviour. The view represents the information to the user and gets data from the model. Each View is associated to a controller that receives input as events (eg. mouse clicks) and proceeds the required services on the model and updates to the view.  5. Structure  The model contains not only the data but also the applications business logic, its functional core. The model has a responsibility to notify dependent components about changes.  Each view defines an update procedure that is activated by the change-propagation mechanism. There is a one-to-one relationship between views and their controllers.  The controller accepts user input as events. How these Events are defined depends on the user interface platform. The behaviour of a controller depends on the state of the model.  https://wiki.ifs.hsr.ch/APF/files/observer.png  Views and Controllers are registered with the model change propagation mechanism. In other words, they are subscribed observers.  6. Dynamics   * Scenario 1: User input results in changes to the model which then notifies all its observers with the updated data. * Scenario 2: The initialization code is usually located in a main program. First the model is created, then the views and finally the controllers.   7. Advantages   * Multiple views of the same model * Synchronized views * Pluggable views and controllers * Exchangeability of look and feel * Framework potential (e.g. Symfony, ASP.NET MVC)   8. Disadvantages / Problems   * Increased complexity * Potential for excessive number of updates * Intimate connection between view and controller (hinders reuse) * Close coupling of views and controllers to a model (changes of the models interface break view and controllers. Apply Command Processor pattern to work around this issue. * Inefficiency of data access in view * Inevitability of change to view and controller when porting. * Difficulty of using MVC with modern user interface tools. | | |
| **Presentation Abstraction Control (PAC)**  **antworten kontrollieren/ergänzen!** | The Presentation-Abstraction-Control Architectural Pattern defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.  1. Context  Development of an interactive application with the help of agents.  2. Problem  In an interactive system, there are many different agents involved. These agents can be aligned horizontally and/or vertically in an architecture. Each agent has a variety of specific tasks (accepting user input, displaying data etc.). All agents together provide the system functionality.  The following forces affect the solution.   * Agents often maintain their own state and data. They need a mechanism for exchanging data, messages and events. * Interactive agents provide their own user interface (entering data using keyboard, manipulation of graphical objects using mouse/touch events etc.) * UI is prone to change. Changing or extending a particular agent should not affect others.   3. Solution  Structure the interactive application as a **tree-like hierarchy** of PAC agents.  There should be one top-level agent, several intermediate-level agents, and even more bottom-level agents. Every agent consists of three components:   * Presentation provides the visible behaviour of the agent. * Abstraction maintains the data model and provides functionality that operates on this data. * Control connects the presentation and abstraction components and provides functionality to communicate with other agents. (acts as adapter and mediator)   The whole hierarchy reflects transitive dependencies between agents. Each agent depends on all higher-level agents.  https://wiki.ifs.hsr.ch/APF/files/solution2.PNG  4. Structure  **Top-level Agent**  It represents the root of the tree. The main responsibility of the top-level agent is to provide global data model of the software. The data within abstraction component is media-independent (eg. size in meters, not in pixels).  The presentation component of the top-level agent often has few responsibilities. It may include common UI to the whole application that cannot be sent to subtasks. e.g. Menu bars, dialog box, close-button, etc.  The control component of the top-level agent has the following three responsibilities.   * Allows lower-level agents to access and manipulate the global data model. * Coordinates the connection between the top-level agent and lower-level agents. * Maintains information about the user interaction with the system.   **Bottom-level Agent**  Represents a leaf node. Bottom-level PAC agents represent a specific semantic concept of the application domain. The concept should be the smallest units a user can manipulate (eg. bar and pie charts).  The presentation component presents a specific view of the semantic concept, and provides functions to all possible user actions (mouse clicks / touch events, key presses etc.).  The abstraction component maintains agent-specific data. Unlike the abstraction component of the top-level agent, no other agents depend on this data.  The control component maintains consistency between the abstraction and presentation components. It serves as an adapter and performs both interface and data adaptation. In this way, direct dependencies between them are avoided. Furthermore, it communicates with high-level agents to exchange events and data.  **Intermediate-Level Agent**  Represents a subtree, consisting of other bottom-level PAC agents. It may contain several views of the same data. e.g. a floor plan in a CAD, with many items in it. Or a game character with many body parts which define their own behaviour.  Hence Intermediate-level agents can fulfil two different roles:   * composition: It groups all the different complex graphic agents to form a composite graphical object. * coordination: It maintains consistency between lower-level agents (eg. coordinates multiple views of the same data).   5. Dynamics   * Scenario 1: When opening a view the view coordinator instantiates the desired presentation which retrieves the data from the top-level PAC agent, finaly displays the all bottom-level presentations. * Scenario 2: When entering data the spreadsheet agent forwards the data to the top-level PAC agent. The top-level PAC agent asks its control component to update all agents that depend on the new data. Top-level PAC agent notifiers view coordinator. View coordinator forwards change notification to all view PAC agents.   6. Advantages   * Separation of concerns * Support for change and extension, changes do not affect other agents and new agents can be easily integrated. * Support for multi-tasking, PAC agents can be distributed easily to different threads, processes or machines.   7. Disadvantages / Problems   * Increased system complexity, when every semantic concept within a application has its own PAC agent. * Complex control component, in a PAC system the control components are the communication mediators, complex to implement. * Lower Efficiency, a lot of overhead in the communication between PAC agents. * Lower Applicability, the smaller the atomic semantic concepts of an application are, and the greater the similarity of ther user interfaces, the less applicable this pattern is.   8. Examples / Usages   * Classic use in a air traffic control system * Used for Games * Used for systems with a lot of changes in Abstraction(Model) | | |
| **Microkernel** | This pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. It is also serving as a socket for plugging in such extensions and coordinating their collaboration.  1. Context  The development of several applications that use similar programming interfaces that build on the same core functionality.  2. Problem  Imagine a software that should run on different machines with different hardware, for instance, an operating system which has a lifetime of about 10 years. This kind of software needs to be adaptive for new technologies in software and hardware. And it should be portable, extensible and adaptable to allow easy integration of emerging technologies. This kind of application platform should also be able to emulate other application platforms that belong to the same application domain, e.g. our new Operating System should also be able to run applications written for Microsoft Windows. This also means that the applications should support different but similar application platforms. One more problem that appears: as the core is the exclusive resource for the applications, it should be of minimal memory size and consume as little processing power as possible.  3. Solution  Encapsulate the fundamental services of your application platform in a microkernel component.  The microkernel   * provides core mechanism. * encapsulates system dependencies. * enables the communication between different components running in separate processes. * is responsible for maintaining system-wide resources such as files or processes * provides interfaces that enable other components to access its functionality.   4. Structure  The Microkernel Pattern defines five kinds of participating components:  **Microkernel**  Represents the main component of the pattern. Other components build on all or on some of these services by using the microkernels interfaces. There are parts in the Microkernel which are invisible from outside. e.g. hardware dependent parts. Clients therefore only see particular views. The MK is also responsible for process- and file-maintaining and coordinates/controls the access to resources. These kinds of services (called mechanisms) serve as the fundament on which policies (complex functions) are constructed.  **Internal Servers (z.B. Geräte Treiber)**  Is a separate component that offers additional functionalities to the MK. The functionalities get invoked by the MK via service requests. Therefore, the Internal Servers can encapsulate dependencies on the underlying hardware or software system. e.g. Drivers for a specific graphic card. As the MK should be lean in resources and execution time, the Internal Servers implement the more complex services which are activated by the MK only when necessary. Internal Servers are only accessible by MK component.  **External Servers**  An ES is also called a personality. It is a component that uses the MK to implement its own layer of abstraction laying on top of the atomic services of the MK. Different ES implement different policies for specific application domains. External Servers expose their functionalities, like the microkernel itself, through Interfaces. Each ES runs in a separate process and it receives application requests using the communication facilities provided by the MK. After processing the request, it also replies to the clients. ESs fully rely on the mechanisms of the MK, so the ES need to access the MK's interfaces.  **Clients**  Is an application that is associated with exactly one External Server and uses their Interfaces. This means that the connection from client to its corresponding server is hardcoded. This leads to bad exchangeability and if the external server changes its behaviour upon application platforms, the client needs to change too. Therefore, it is better to implement interfaces between each client and its external server... | | |
| **Adapters**  An adapter is the corresponding interface between the client and its external server. Its purpose is, that if an external server changes upon its application platform, the adapter mimics the programming interfaces of that platform. Therefore, the clients do not need to be modified. | https://wiki.ifs.hsr.ch/APF/files/layered-microkernel.png | |
| 5. Dynamics   * Scenario 1: Behaviour when a client calls a service of its external server. https://wiki.ifs.hsr.ch/APF/files/MicrokernelDynamics1.JPG * Scenario 2: External server requests a service that is provided by an internal server.  https://wiki.ifs.hsr.ch/APF/files/MicrokernelDynamics2.JPG   6. Variants   * Microkernel System with indirect Client-Server connections. In this variant a client that wants to send a request of message to an external server asks the microkernel for a communication channel, the microkernel is used as a message backbone. * Distributed Microkernel System, in this variant every machine uses its own microkernel implementation. Additional services for communication with each other must be provided. From the user's viewpoint, the whole system appears as a single Microkernel System.   7. Known Uses  The classic use for microkernels are operating systems like Mach, Amoeba or Windows NT. MKDE system introduces an architecture for database engines that follows the microkernel pattern.  8. Advantages   * The Microkernel offers a high degree of Portability. * One of the biggest strengths is its Flexibility and Extensibility. * Separation of policy and mechanism. Microkernel component provides all the mechanisms necessary to enable external servers to implement their policies. * Scalability: microkernel could be distributed which makes it easy to scale. * Reliability: microkernels are high available and fault tolerant which makes them more reliable. * Transparency: in a distributed system components can access each other without knowing their location.   9. Disadvantages   * Performance, the microkernel system supports different views, which results in worse performance in the most cases. * High Complexity of design and implementation, it is not a trivial task to develop a microkernel based system. | | |
| **Broker** | The Broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication. i.e. forwarding requests to the server and transmitting its answers and exceptions back to the caller.  1. Problem  You decided to design your application as a set of decoupled and interoperating components, rather than a monolithic application. this results in greater flexibility, maintainability and changeability. By partitioning functionality into independent components the system becomes potentialliy distributable and scalable. But if components handle communication themselves the resulting system faces several dependencies and limitations. e.g. clients need to know the location of the servers, the system is depending on the communication mechanism, etc. And for the developer, there should be no difference between developing software for centralized systems and developing for distributed ones.  2. Solution  Servers register themselves with the broker, and make their services available to clients through method interfaces. Clients access the functionality of servers by sending requests via the broker. The broker is responsible for locating the server, forwarding the requests and transmit the result or exceptions back to the client.  By using the Broker pattern, an application can access distributed services simply by sending message calls to the appropriate object.  The Broker pattern reduces the complexity, because it makes distribution transparent to the developer. It extends object models from single applications to distributed applications consisting of decoupled components that can run on heterogeneous machines and can be written in different languages.  ildergebnis für broker pattern  3. Structure  The Broker Pattern defines six kinds of participating components:  **Clients**  Clients are applications that access the services of at least one server. To call remote services, clients forward requests to the broker. After an operation has executed they receive responses or exceptions from the broker.  **Servers**  A server implements objects that expose their functionality through interfaces that consist of operations and attributes. These interfaces are made available either through an interface definition language (IDL) or through a binary standard. There are two kinds of Servers:   * Servers offering common services to many application domains. * Servers implementing specific functionality for a single application domain or task.   **Brokers**  A broker is a messenger that is responsible for the transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client. A broker must have some means of locating the receiver of a request based on its unique system identifier. A broker offers APIs to clients and servers that include operations for registering servers and for invoking server methods.  **Bridges**  Bridges are optional components used for hiding implementation details when two brokers interoperate. Suppose a Broker system runs on a heterogeneous network. If requests are transmitted over the network, different brokers have to communicate independently of the different network and operating systems in use. A bridge builds a layer that encapsulates all these system-specific details.  **Client-side proxies**  Client-side proxies represent a layer between clients and the broker. This additional layer provides transparency, in that a remote object appears to the client as a local one. In many cases, client-side proxies translate the object model specified as part of the Broker architectural pattern to the object model of the programming language used to implement the client.  **Server-side proxies**  Server-side proxies are generally analogous to Client-side proxies. The difference is that they are responsible for receiving requests, unpacking incoming messages, unmarshalling the parameters, and calling the appropriate service. They are used in addition for marshalling results and exceptions before sending them to the client.  The following diagram shows the objects involved in a Broker system:  https://wiki.ifs.hsr.ch/APF/files/BrokerStrucutreOverview.JPG  4. Dynamics   * Scenario 1: Behavior when a server registers itself with the local broker component.  https://wiki.ifs.hsr.ch/APF/files/BrokerDynamics1.JPG * Scenario 2: Behaviour when a client sends a request to a local server, synchronous invocation (clients blocks until it gets a response from the server). Brokers may also support asynchronous invocations.  https://wiki.ifs.hsr.ch/APF/files/BrokerDynamics2.JPG * Scenario 3: Interaction of different brokers via bridge components https://wiki.ifs.hsr.ch/APF/files/BrokerDynamics3.JPG   5. Variants  **Direct Communication Broker System**  In this variant clients can communicate with servers directly. The broker tells the clients which communication channel the server provides.  **Message Passing Broker System**  In this Variant servers use the type of a message to determine what the clients must do, rather than offering services that clients can invoke.  **Trader System**  The request will not be forwarded to a specific server, it will be forwarded to all servers which implement a specific service.  **Other Variants:**   * Callback Broker System * Adapter Broker System   6. Advantages   * Location Transparency: Clients do not need to know the location of the servers as the broker is responsible for that. * Changeability and extensibility of components * Portability of a Broker system: The Broker hides system-specific details from clients and servers at the lower-most layers. Thus, it is very easy to port these lower-most layers instead of completely porting the broker component. * Interoperability between different Broker systems: If a common protocol is used for exchanging messages, different broker systems can interoperate. * Reusability: When building new client applications, you can often base the functionality of your application on existing services.   7. Disadvantages   * Restricted efficiency: Applications are usually slower, because Broker architectural pattern uses indirection layers such as APIs, proxies and bridges. * Lower fault tolerance: If a server or a broker fails during program execution, all the dependent applications are unable to continue successfully. You can increase the reliability through replication of components.   8. Neutral Consequences   * Testing and Debugging: A client application is more robust and easier itself to test. However, debugging and testing a Broker system is tedious job because of many components involved. | | |
| **Whole part** | The Whole-Part design pattern helps with the aggregation of components that together form a semantic unit. An aggregate component, the *Whole*, encapsulates its constituent components, the *Parts*, organizes their collaboration, and provides a common interface to its functionality. Direct access to the Parts is not possible.  1. Problem and Forces   * The behaviour and properties of the individual parts depends on the whole. * A complex object should be decomposed into smaller parts to support reusability and changeability. * Clients should see the complex aggregate object as an atomic whole. There is no direct access to individual parts of them.   2. Solution   * Define a Whole component that encapsulates smaller Parts and prevents clients from accessing these constituent parts directly.   3. Structure  https://wiki.ifs.hsr.ch/APF/files/whole-part.png  **Whole**: represents an aggregation of smaller objects (e.g. GUI Element: Window, Elements). It uses the functionality of *Part* objects for implementing services.  **Part**: Each part is embedded in exactly one *Whole*. Two or more *Wholes* cannot share the same *Part*.  There are 3 different types of relationships that can exist between a *Whole* and a *Part*:   * *assembly-parts* (e.g. molecule-atoms): All parts are tightly integrated to the assembly. The amount and type of subassemblies are predefined. * *container-contents* (e.g. Dependency Injection Container): The contents are less tightly coupled and may be dynamically added or removed. * *collection-members* (e.g. Iterable Sets): It helps to group similar objects. The collection provided functionality to iterate over its members and to perform operations on each of them.   4. Dynamics | | |
| **https://wiki.ifs.hsr.ch/APF/files/whole-part-dynamics.png** | | https://wiki.ifs.hsr.ch/APF/files/whole-part-dynamics_2.png | |
|  | 5. Variants   * Shared Parts: Allows Parts to belong to more than one Whole. * Assembly-Parts: Assembly is a Whole object that represents an assembly of objects. * Container-Contents: Container is responsible for maintaining differing contents. * Collection-Members: Specialization of Container-Contents. In this variant, all members objects (Parts) have the same type. * Composite: Wholes and Parts can be treated uniformly - i.e. in which both implement the same abstract interface. For instance, a folder (Whole) and a file (Part) both implements FileSystemObject Interface.   6. Benefits:   * Changeability of Parts: Since parts are not visible to clients, they can be easily exchanged. * Separation of concerns: Each part has its own responsibility, the Whole manages the aggregated behaviour. * Reusability: Wholes can be reused, because their coupling to parts is not transparent to clients.   7. Liabilities   * Lower efficiency through indirection. * Complexity of decomposition into parts: This is relevant if the behaviour of certain parts depends not only on their whole, but also on the behaviour of other parts. | | |
| **Master Slave** | The design pattern Master-Slave supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a final result from the slaves' results.  1. Problem  Let's take the Travelling Salesman Problem where the Traveling Salesman wants to travel through all the states in the US. There are 6.08\* 10^62 different trips. So we want to parallelize the calculation. But with this 'Divide and Conquer' principle, there arise several forces:   * Clients should not be aware that the calculation runs parallel * Clients and subtasks should not be responsible for dividing the calculation and for collecting and unifying the results * It is helpful to use semantically-identical calculation implementations for the subtasks * Processing of subtasks needs coordination   2. Solution  Introduce a coordination instance (a master) between clients and the processing of the individual subtasks.  A master component divides work into subtasks, delegates these subtasks to several independent but semantically-identical slave components, and computes a final result from the partial results.  This ensures the general principles: fault tolerance, parallel computation and computational accuracy.  **Fault Tolerance**  The execution of a service is delegated to several replicated implementations which, in case of a fail, can be detected and handled.  **Parallel computing**  A complex task is divided into subtasks and the results are being combined again.  **Computational Accuracy**  The calculation is divided into subtasks, inaccurate results can be detected and handled.  In general: The length of the trips are calculated in parallel and implement it as a slave. Each slave takes a number of trips, and calculates and returns the shortest trip found. The master determines the a priori the number of slaves that are instantiated, specifies how many trips a slave takes, instantiates the slaves and selects the shortest trip from all returned trips.  3. Structure  Provide all slaves with a common interface. Let clients communicate only with the master. This means that there is one master and it has at least two slaves:  https://wiki.ifs.hsr.ch/APF/files/master-slave-structure.png  4. Dynamics  The client calls a service from the master. The master splits the work and for each subtask it calls a separate slave. These run the subtask and return its result to the master. The master combines the results and returns them to the client.  https://wiki.ifs.hsr.ch/APF/files/master-slave-dynamics.png  5. Variants  There are three application areas for the Master-Slave pattern, and three Slave variants.  5.1 Master-Slave Variants  Master-Slave...   * ... for fault tolerance: Master delegates the execution of a service to X Slaves. As long as the first slave terminates, the result is returned to the client. If one slave fails, the other redundant slaves help to provide a valid result to the client. The master may use time-outs to detect slave failure. If all slaves fail, the master raises an exception or returns an exceptional value to the client. However, if the master itself fails, the whole system will get the state failed. * ... for parallel computation: Master divided a complex task into X identical sub-tasks, each of which are executed in parallel by a separate slave. The master builds the final result by combining the results obtained from the slaves. * ... for computational accuracy: Master delegates the execution of a service to at least 3 different slaves and waits for all slaves to complete. Then the master votes on their results to detect and handle inaccuracies. The voting may follow different strategies, such as the most frequent result, the average of all results or use of an exceptional value if all slaves produce different values.   https://wiki.ifs.hsr.ch/APF/files/master-slave-variatns.png  5.2 Slave Variants  Slaves...   * ... as Processes: To handle slaves located in separate processes, the master includes a top component that keeps track of all slaves working for the master. Additionally, remote proxies are used to represent each slave in the master process in order to keep the master and top component independent of the physical location. Furthermore, the Forwarder-Receiver or Client-Dispatcher Pattern can be applied to implement inter-process communication. * ... as Threads: The master creates the threads, launches the slaves, and waits for all threads to complete before continuing with its own computation. * Master-Slaves with slave coordination: Sometimes the computation of a slave depend on other slaves. In this case the computation of all slaves must be regularly suspended, so that each slave can coordinate itself with its dependent slaves. After that the slaves resume their individual computation.   https://wiki.ifs.hsr.ch/APF/files/rsz_slave-variants.png | | |
| **Proxy** | The Proxy design pattern makes the clients of a component communicate with a representative rather than to the component itself. Introducing such a placeholder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorized access.  1. Problem  It is often inappropriate to access a component directly. We do not want to hard-code its physical location into clients, and direct and unrestricted access to the component may be inefficient or even insecure. Additionally, we have to balance following forces:   * Accessing the component should be run-time efficient, cost effective and safe for client and component. * Access to the component should be transparent and simple. Client should not have to change its calling behaviour and syntax.   2. Solution  Let a client communicate with a representative rather than the component itself. This representative, called a *proxy*, offers the interface of the component but performs additional pre- and post-processing such as access-control checking or making read-only copies of the original.   * Client should be aware of possible performance or financial penalties for accessing remote clients.   3. Structure  The original implements a particular service. The client is responsible for a specific task. To do its job, it invokes the functionality of the original in an indirect way by accessing the *proxy*. The client does not have to change its calling behaviour and syntax.  The proxy offers the same interface as the original, and ensures correct access to the original. To achieve this the proxy maintains a reference to the original it represents.  The abstract original provides the interface implemented by the proxy and the original. Both the proxy and the original inherit form the abstract original. Clients code against this interface when accessing the original.  https://wiki.ifs.hsr.ch/APF/files/ProxyStucture.PNG  4. Dynamics   * While working on its task the client asks the proxy to carry out a service. * The proxy receives the incoming service request and pre-processes it (Looking up address of original, checking local cache). * If the proxy has to consult the original to fulfil the request, it forwards the request. * The original accepts the request and fulfils it. It sends the response back to the proxy. * The proxy receives the response and post-processes it (caching the result, calling destructor of original, releasing a lock on resource). Afterwards it sends the result to the client.   https://wiki.ifs.hsr.ch/APF/files/ProxyDynamics.PNG  5. Variants   * Remote Proxy: provides a local represent of an object in different address * Virtual Proxy: Processing or loading a component is costly, while partial information about the component may be sufficient. * Protection Proxy / Access Control Proxy: Components must be protected from unauthorized access. * Cache Proxy: Multiple local clients can share results from remote components. * Synchronization Proxy: Multiple simultaneous accesses to a component must be synchronized. * Counting Proxy / Smart Reference Proxy: Accidental deletion of components must be prevented or usage statistics collected. * Firewall Proxy: Local clients should be protected from the outside world ("bad" clients).   6. Similar Patterns  **Adapter vs. Proxy**  Adapter offers different Interface than Adaptee.  Proxy offers same Interface as Real Subject  **Facade vs. Proxy**  Facade: simplified interface to a larger complex system.  Proxy: Reference to a single object  **Decorator vs. Proxy**  Decorator: Adding more responsibilities/behaviour to object while decorating.  Proxy: Controlling access to the original object | | |
| **View Handler** | The View Handler design pattern helps to manage all views that a software system provides. A view handler component allows clients to open, manipulate and dispose of views. It also coordinates dependencies between views and organizes the update of those.  1. Problem  A software system that supports multiple views often rely on additional functionality for managing these views. As a user, editing any view, I want to have the update propagated automatically to the other views. Therefore, the following problems appear:   * Managing multiple views should be easy from the user's perspective * Managing multiple views should also be easy for client components within the system * Implementation of Views should not depend on each other or be mixed with code * Views may vary and additional types of views may be added during the lifetime of the system   2. Solution  The management of the views should be separated from the presentation. This is the idea of the separation of presentation and functionality.   * A view handler component manages all views that the software system provides (opening, coordinating, closing, handling). * The View components are responsible for the presentation of all Data. * The Supplier provides the Data, that should be presented.   3. Structure    **View Handler**  View Handler is the central component and is responsible for opening new views requested by the clients.  If the requested view is open already, the view handler brings this open view to the foreground.  If the requested view is open but iconized, the view handler tells the view to display itself full size.  The view handler also offers functions for closing views.  Further features: tile all views, clone views, refresh views if model is changed  **Abstract View**  An abstract view component defines an interface that is common to all views.  The view handler uses this interface for creating, coordinating, and closing views.  The platform underlying the system uses the interface to execute user events, such as resizing window.  In addition to that, the interface of the abstract view must offer a corresponding function for all possible operations that can be performed on a view.  **Specific View**  Specific view implements the abstract view interface.  In addition, each view implements its own display function.  It retrieves data from the view's suppliers, prepares this data for display and presents them to the user.  **Supplier**  Supplier provides the data that is displayed by view components.  It also offers an interface that allows views to change data. They notify dependent views via view handler about changes to their internal state.  4. Dynamics  The following two Scenarios assume, that each view is displayed in its own window.  4.1 Scenario I   1. The client calls the view handler to open a particular view 2. View handler instantiates and initializes the desired view. The view Registers to the change-propagation mechanism of its supplier 3. View handler adds the view to its internal list 4. View handler calls the view and requests to open itself. The view opens a new window, retrieves data from its supplier, prepares this data and presents them.   https://wiki.ifs.hsr.ch/APF/files/ViewHandler-Scenario1-v2.png  4.2 Scenario II  For simplicity, we assume that only two views are open.   1. Client calls the command to tile all open windows. 2. for every open view, the view handler calculates the new size and position, and calls its resize and move procedures 3. Each view changes its position and size, sets the corresponding clipping area and refreshes the image it displays to the user.   https://wiki.ifs.hsr.ch/APF/files/ViewHandler-Scenario2.png  5. Consequences  5.1 Benefits   * Uniform Handling of Views: All views share a common interface. * Extensibility and changeability of views: Due to abstract base, additional views and changes to views can easily be implemented * Application-specific view coordination: Since the abstract base class, it is possible to implement specific view coordination strategies   5.2 Liabilities   * Restricted applicability * Efficiency | | |
| **Forwarder Receiver** | The Forwarder-Receiver design pattern provides transparent inter-process communication for software systems with a peer-to-peer interaction model. It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms.  1. Problem  The common way to build distributed applications is to make use of available low-level mechanisms for inter-process communication such as TCP/IP, sockets or message queues. These are very efficient compared to higher-level mechanisms but they often introduce dependencies on the underlying operating system and network protocols. Using IPC restricts the portability.  The Forwarder-Receiver pattern is useful when you need to balance following forces:   * System should allow exchangeability of the communication mechanisms. * Cooperation of components follows a peer-to-peer model, in which a sender only needs to know the names of it receivers. * Communication between peers should not have a major impact on performance.   2. Solution  Distributed peers collaborate to solve a particular problem. A peer may act as a client, requesting services, as a server, providing services or both. The details of the underlying IPC mechanism for sending or receiving messages are hidden from the peers by encapsulating all system-specific functionality into separate components. Examples of such functionality are the mapping of names to physical locations, the establishment of communication channels, or the marshalling (similar to serialization) and unmarshalling (cf. deserialization) of messages.  3. Structure  The Forwarder-Receiver design pattern consists of three components:   * forwarders * receivers * peers   https://wiki.ifs.hsr.ch/APF/files/structure.PNG  **Peer**  Peers are responsible for application tasks. To carry these out, they often need to communicate with other peers, which are either on different processes or other machines. Each peer knows the name of the other peers. It uses a forwarder to send messages and a receiver to receive messages from peers. Peers continuously monitor network events and resources, and listen to any other agent to exchange information and requests.  **Forwarder**  Sends messages across process boundaries. Provides an interface that is an abstraction of a particular inter-process communication (IPC) mechanism, and includes functionality for marshalling and delivery of messages. It also contains a DNS system. When a forwarder sends a message to a remote peer, it specifies the name of its own peer in order for the remote peer to be able to send a response to the originator. The forwarder sends messages without introducing any dependency on the underlying IPC mechanism.  **Receiver**  Receivers are responsible for receiving messages. A receiver offers a general interface that is an abstraction of a particular IPC mechanism. It includes functionality for receiving and unmarshalling messages.  4. Dynamics  https://wiki.ifs.hsr.ch/APF/files/forwarder-receiver-dynamics.png  In this example two peers (P1, P2) communicate with each other.   1. P1 requests a service from a remote peer P2. For this purpose, it sends the request to its forwarder Forw1 and specifies the name of the recipient. 2. Forw1 determines the physical location of the remote peer and marshals the message 3. Forw1 delivers the message to the remote receiver Recv2 4. At some earlier time P2 has requested its receiver Recv2 to wait for an incoming request. Now, Recv2 receives the message arriving from Forw1 5. Recv2 unmarshalls the message and forwards it to its peer P2 6. Meanwhile, P1 calls its receiver Recv1 to wait for a response 7. P2 performs the requested service and sends the result and the name of the recipient to its forwarder Forw2. 8. The forwarder marshalls the result and delivers it t Recv1 9. Recv1 receives the response, unmarshalls it and delivers it to P1   5. Variants  Forwarder Receiver without name-to-address mapping  6. Consequences  6.1 Advantages   * Efficient inter-process communication. A forwarder does not have to locate remote components. * Encapsulation of IPC facilities. All dependencies on concrete IPC facilities are encapsulated within the forwarders and receivers. Change of the underlying IPC mechanism does not affect other components.   6.2 Disadvantages   * No support for flexible re-configuration of components. Forwarder-Receiver systems are hard to adapt if the distribution of peers may change at run-time. Problem can be solved by adding a central dispatcher component. | | |
| **Client Dispatcher Server** | The Client-Dispatcher-Server design pattern introduces an intermediate layer between clients and servers, the dispatcher component. It provides location transparency by means of a name service, and hides the details of the establishment of the communication connection between clients and servers.  1. Context  A software system integrating a set of distributed servers, with the servers running locally or distributed over a network.  2. Problem  When as software system uses servers distributed over a network, it must know the location. If server location changes, client should also adapt it.  Forces:   * A component should be able to use a service independent of its location. * The code implementation of the service and connection establishment should be separated.   3. Solution  Provide a dispatcher component to act as an intermediate layer between clients and servers.  The dispatcher implements a name service that allows clients to refer to servers by names instead of physical locations -> location transparency.  4. Structure  https://wiki.ifs.hsr.ch/APF/files/client-dispatcher-server.PNG   * Client - Before sending a request to a server, the client asks the dispatcher for a communication channel. The client uses this channel to access operations offered by servers in order to carry out its processing tasks. * Server - provides services to the clients. It either registers itself or is registered with the dispatcher by its name and address. * Dispatcher - establishes communication channels between clients and servers. To do this, it first takes the server name and maps this name to the physical location of the server. When the connection is established, it returns a communication handle to the client. Moreover, dispatcher implements functions for registering and location servers.   5. Dynamics  https://wiki.ifs.hsr.ch/APF/files/ClientDispatcherDynamics.JPG  6. Variants   * Distributed Dispatchers * Client-Dispatcher-Server with communication managed by clients * Client-Dispatcher-Server with heterogeneous communication * Client-Dispatcher-Service   7. Consequences  7.1 Benefits   * Exchangeability of servers * Location and migration transparency * Reconfiguration * Fault tolerance   7.2 Liabilities   * Lower efficiency through indirection and explicit connection establishment * Sensitivity to change in the interfaces of the dispatcher component | | |
| **Counted Pointer / CountingHandle** | Simplify the lifetime management of shared heap objects by introducing handle objects that act as the references to the heap object, tracking its usage | | |
| The Counted Pointer idiom makes memory management of dynamically-allocated shared objects in C++ easier. It introduces a reference counter to a body class that is updated by handle objects. Clients access body class objects only through handles via the overloaded operator->(). 🡪 std::shared\_ptr<T> | | |
| /Users/Michi/Desktop/Bildschirmfoto 2018-01-05 um 13.06.03.png | | |

|  |  |
| --- | --- |
| **POSA 2** | |
|  | |
| **Wrapper Facade** | 1. Zusammenfassung  Wrapper Facade kapselt die Funktionalität von bestehenden nicht objekt-orientierten APIs in objekt-orientierte Interfaces/Objekte. Diese sind besser wartbar und einfacher portierbar.  2. Kontext  Applikationen, welche Services von bestehenden, nicht oo APIs verwenden und zudem Erweiterbar und Wartbar sein müssen.  3. Problem  - Objekt-Orientierung und die damit verbundenen Highlevel-Features reduzieren den Entwicklungs- & Wartungsaufwand. Low-Level API’s können davon keinen Gebrauch machen.  - Einfache Portierbarkeit erhöht die Einsatzmöglichkeiten und Attraktivität einer API  - Um Wartungsaufwand zu minimieren werden Kompiler-Switches eingesetzt. Diese erhöhen aber die Komplexität der Applikation und verstreuen die Funktionalität an mehrere Orte  - Low-Level Sprachen fehlen die Möglichkeit, Funktionalität zu strukturieren und so die Kohäsion zu erhöhen, was wiederum die Komplexität verringern würde.  - Ohne Schnittstelle müssen solche API’s direkt angesprochen werden, was dann den Aufwand aufgrund der genannten Probleme erhöht  4. Lösung  Erstelle pro zusammengehörigen API-Funktionen / Daten eine Wrapper-Facade in einer High-Level Sprache, die die Funktionen kapselt und durch objekt-orientierte Features anreichert.  5. Struktur  6. Implementation  Folgendes Vorgehen ist nötig, um Wrapper Facade zu implementieren:  1. Identifiziere Zusammenhänge zwischen bestehenden low-level APIs  2. Gruppiere zusammenhängende Funktionen in Wrapper Facade Klassen  1. Erstelle zusammenhängende Klassen  2. Vereinige mehrere individuelle Funktionen in einer einzigen Methode  3. Wenn möglich: Automatisiere Erstellen/Löschen  4. Entscheide wo plattform-spezifische Funktionalität gekappselt werden soll  3. Entscheide, ob kontrollierter Zugriff auf implementations Details erlaubt werden soll  4. Entwickle ein adäquates Error-Handling  5. Definiere verwandte Helferklassen (optional)  7. Known Uses  MFC: Microsoft Foundation Classes  Kapselt low-level C API Call für GUI-Komponenten (MS Document View)  JVM: Java Virtual Machine und diverse Java class Libraries (z.B AWT/Swing)  Kapselt native Betriebssystem Aufrufe und GUI APIs  8. Vorteile  - Erweitert die Lowlevel-API Calls um Highlevel Features wie z.B. Typesafety und senkt so die Fehleranfälligkeit  - Erhöht die Kohäsion und damit die Verständlichkeit des Codes  - Portierbarkeit kann durch logische Struktur (z.B. Vererbung) ersetzt werden, anstatt durch physische Struktur (compiler-switch)  - Die Gruppierung macht es einfacher, die Wrapper-Facade an anderen Stellen wiederzuverwenden  - Viele Fehlerquellen können minimiert werden durch Features wie Konstruktoren/Destruktoren oder Exceptions  9. Nachteile  - Es kann Funktionalität verloren gehen, wenn man die API nur noch über die Facade anspricht  - Je nach Sprache kann es zu Performance-Verlust kommen  - Es kann nicht jede High-level Sprache für jede Low-Level Sprache verwendet werden. Die Facade ist in in ihrer Sprache eingeschränkt, kann z.B. nicht in Java geschrieben sein.  10. Verwandte Patterns  - Facade: Facade hat zum Ziel, ein vereinheitlichted Interface anzubieten, welches Client Zugriff auf Subsysteme vereinfacht. Facade versteckt also komplexe Klassen Abhängigkeiten hinter einem einfachen API. Wrapper Facade hingegen versteckt komplexe Klassen Abhängigkeiten und Datenstrukturen hinter mächtigen objekt-orientierten Klassen.  - Bridge: Bridge hat zum Ziel, Abstraktion von Implementation zu trennen, so dass sich beide unabhängig verändern können, Erreicht wird dieses Ziel mit Hilfe von Polymorphismus. Wrapper Facade hat ein ähnliches Ziel: minimizing the overhead of indirection and polymorphism.  - Adapter: Adapter hat zum Ziel, das Interface einer Klasse umzuwandeln in eines, welches von einem Client erwartet wird. Wrapper Facades spielen eine ähnliche Rolle, indem sie ein objekt-orientiertes Interface exportieren, welches plattform unabhängig ist.  - Decorater: Decorater hat zum Ziel, ein Objekt dynamisch zu erweitern, indem Zuständigkeiten transparent agehängt werden. Im Gegensatz dazu kapselt Wrapper Facade lower-level Funktionen und Daten statisch mit objekt-orientierten Interfaces.  - Generell kann gesagt werden, dass Wrapper Facade anstelle dieser GoF Patterns verwendet werden soll, wenn:  - lower-level, nicht objekt-orientierte APIs bestehen  - Effizienz wichtiger ist als dynamische Erweiterbarkeit |
| **Inter-ceptor** | 1. Zusammenfassung  Das Interceptor Architektur-Pattern erlaubt zusätzliche Services zu einem existierenden Framework hinzuzufügen, ohne dieses verändern zu müssen.  2. Kontext  Entwickeln eines Frameworks, welches dynamisch und transparent erweitert werden kann.  3. Problem  - Frameworks müssen oft mehr Services anbieten, als im Kern zur Verfügung stehen.  - Speziell Black-Box Frameworks können oft nicht so einfach erweitert werden  - Architekturentwickler möchten nicht die Core Architektur mit vielen Features überladen, sondern möglichst viel wiederverwenden.  - Existierende Software-Komponenten sollen so wenig wie möglich angepasst werden müssen.  - Die Integration eines neuen Service in ein Framework sollte die existierenden Komponenten des Frameworks nicht beeinflussen  4. Lösung  - Man erlaubt externen Diensten (Applications) das Framework zu erweitern, in dem man Schnittstellen anbietet, welche vom Framework bei bestimmten Events aufgerufen werden.  - Über vordefinierte Interceptor Callback Interfaces werden Events an die Applications (externe Services) weitergereicht.  - Die Framework-Implementation soll geöffnet werden, damit eine Application ein bestimmtes Framework-Verhalten überwachen und steuern kann.  - Wenn ein bestimmter Event im Framework auftritt, werden alle beim Dispatcher registrierten Interceptors benachrichtigt. (Über eine Callback-Funktion)  5. Struktur  Das Pattern besteht aus fünf Aktoren:  **Dispatcher**  - Vermittelt zwischen Framework und Interceptor  - Beinhaltet Methoden für die Registration und Deregistration von konkreten Interceptoren.  - Definiert ein Interface, welches vom konkrenten Framework aufgerufen wird, wenn entsprechende Events ausgelöst wurden.  - Typischerweise wird ein Dispatcher pro abstrakter Interceptor erstellt  **(konkreter) Interceptor**  - Enthält Callback Methoden die vom Dispatcher aufgerufen werden  - Implementiert das Interceptor-Interface  **Context Object**  - Das Context Objekt enthält spezifische Informationen zu einem Framework Event, sowie Methoden um den internen Status des Framework anzupassen  - Context Objekte werden von Interceptoren verwendet, um auf interne Aspekte des Frameworks zuzugreifen und diese zu verändern.  - Bietet Getter (Accessor) und Setter (Mutator) für den Zugriff auf interne Aspekte  **Application**  - Nutzt die grundlegenden Funktionen des Frameworks und erweitert diese  **(konkretes) Framework**  - Wird dynamisch erweitert  - Ruft Dispatcher auf bei einem Event  6. Ablauf  1. Die Application instanziiert einen konkreten Interceptor  2.Die Application registriert diesen konkrenten Interceptor beim entsprechenden Dispatcher  1. Typischerweise wird ein Dispatcher pro Interceptor erstellt.  3. Das Framework instanziiert ein event-spezifisches Context Objekt  1. Typischerweise wird pro Framework Event ein Context Objekt erstellt  2. Das Context Objekt enthält spezifische Informationen zum Event, sowie Methoden um den internen Status des Frameworks anzupassen  4. Das Framework notifiziert den entsprechenden Dispatcher über das Ereignis und übergibt ihm das neue Context Objekt.  5. Der Dispatcher iteriert über alle registrierten Interceptoren und ruft dabei deren Callback/Hook Methoden auf. Beim Aufruf wird das Context Objekt als Parameter an den konkreten Interceptor übergeben.  6. Der konkrete Interceptor kann dann den gekapselten Event verarbeiten und allenfalls über die Getter und Setter Methoden das interne Verhalten des Frameworks verändern.  7. Sobald alle registrierten Interceptoren den Prozess verarbeitet haben, geht der normale Betrieb des Frameworks weiter.  7. Implementation  Für die Implementation des Interceptor Patterns sind sieben Schritte nötig:  Internes Verhalten des konkreten Frameworks als State-Machine modellieren: Um die Komplexität zu minimieren, werden oft einzelne Teile des konkreten Frameworks als Sub State Machine modelliert. Mithilfe dieses Models kann einfacher herausgefunden werden, wo und wann ein Event intercepted werden kann.  Identifikation der Interception-Points: Wird in 4 Teilaktivitäten unterteilt:  Identifikation der Statusübergänge, die evtl. nicht für externen Applikation sichtbar sind, aber trotzdem abgefangen werden möchten.  Partitionierung der Interceptor Points in Reader und Writer Sets.  Reader Set: Beinhaltet alle lesenden State Transitions  Writer Set: Beinhlatet alle schreibenden State Transitions  Integration der Interception Points in das State Machine Model: Erstelle einen neuen Interception State (Zwischenschritt) in dem Model.  Partitioniere Interception Points in disjunkte Gruppen. (Interception Group)  Spezifizieren der Context Objekte  Spezifizieren eines Inceptors für jede identifizierte Interception Group (2.4) und für jeden enthaltenen Interception Point eine Callback Methode.  Spezifizieren des Dispatchers für jeden Interceptor  Implementieren des Callback Mechanismus im konkreten Framework mit dem Observer Pattern. Der Mechanismus propagiert Events vom konkreten Framework zu den Dispatchern und von dort zu den registrierten Interceptors.  Implementieren der konkreten Interceptoren  8. Benötigte Patterns  Observer: Für die Event Notifizierung zwischen Framework<>Dispatcher<>Interceptor  Iterator für den internen Callback Mechanismus im Dispatcher  Strategy, damit Applications die Aufruf-Reihenfolge der Interceptors steuern können (FIFO, LIFO)  9. Known Uses  CORBA  Webbrowser (damit Addons Einfluss nehmen können)  10. Vorteile  - Erweiterbarkeit und Flexibilität  - Separation of concerns  - Unterstützung für Monitoring & Kontrolle des Frameworks  - Wiederverwendbarkeit (ein Interceptor kann gut in andere Projekte portiert werden, die das gleiche Framework einsetzen)  - Nachteile ^  Es ist schwer herauszufinden, wo Interceptors überall Sinn machen (Wenn zuviele Interceptors definiert werden, kann das Design oder die Performance darunter leiden)  Wenn keine Timeouts verwendet werden, kann die Applikation blockieren, wenn ein Interceptor blockiert (üblicherweise laufen Interceptors im gleichen Adressraum wie die Applikation)  Bösartige oder fehlerhafte Interceptors können Probleme verursachen.  Gefahr von rekursiven Interceptions (Wenn während einer Interception ein Context-Objekt etwas ändert, das dann wiederum weitere Events auslöst, kann es zu Loops kommen)  11. Verwandte Patterns  - Template Method: Spezifiziert ein Gerüst für einen Algorithmus, wobei gewisse Schritte des Algorithmus variieren können. Die Ausführung dieser varianten Schritte wird an "Hook" Methoden delegiert, welche von Subklassen überschrieben werden können. Die Template Method ist also eine Art von leichtgewichtigem Framework und die "Hook" Methoden eine Art von Interceptors.  - Chain-of-Responsibility: Definiert verschiedene Handler, welche zwischen Sender und Empfänger eingefügt werden können. Dabei werden Requests solange weiter propagiert, bis ein Handler den Request abarbeitet. Im Gegensatz dazu werden beim Interceptor Pattern die Events an alle registrierten konkreten Interceptoren weitergeleitet.  - Proxy: Stellt einen Platzhalter für die Interaktion mit sich selber bereit. Proxies können zwar gebraucht werden, um zusätzliche Funktionalität in ein System zu integrieren, sind aber eingeschränkt auf Objekte, welche bereits im System sichtbar sind. Interceptoren dagegen erlauben es externen Komponenten, auf interne und ansonsten "unsichtbare" Komponenten zuzugreifen.  - Observer und Publisher/Subscriber: Helfen den Zustand von kooperierenden Komponenten zu synchronisieren. Diese Pattern bieten aber nur eine Einweg Kommunikation vom Publisher zum Subscriber und spezifieren so nicht, wie ein Observer/Subscriber auf die Funktionalität des Publishers zugreifen sollen. |
| **Reactor** | **Zusammenfassung**  Eine Applikation empfängt Events echt parallel, z.B. als Server Requests. Sie kann diese aber nur seriell abarbeiten. Der Reactor sorgt dafür, dass Events möglichst effizient seriell in einem Thread bearbeitet werden, ohne diesen zu blockieren.  **Alternative Begriffe** Dispatcher, Notifier.  **Kontext**  Event-driven Software, welche mehrere Service-Requests parallel erhält, diese jedoch synchron und seriell verarbeiteten.  **Problem**  Ein Server muss in der Lage sein, parallelle Requests entgegenzunehmen. Diese Requests gilt es jetzt möglichst effizient abzuarbeiten. Der einfachste Ansatz wäre es, einen Thread pro Request zu erstellen. Allerdings ist das sehr ineffizient, da Requests oft blockieren, z.B. bei Filesystem-calls.  Es benötigt also eine Komponente, die die parallelen Events "demultiplext" und sie so in eine sequentielle Reihenfolge bringt.  Dass Pattern muss den folgenden Anforderungen genügen:   * Es darf keine eintreffenden Events blockieren. * Clients sollen fair behandelt werden * Es soll keine unnötigen Contextswitches oder Datenverschiebungen zwischen CPUs stattfinden (ineffizient) * Einfügen von neuen Services oder Ändern von bestehenden, soll nur wenig Aufwand generieren * Der Applikationscode soll nichts mehr von Multithreading oder Synchronisierung wissen   Das Pattern invertiert den Kontrollfluss (Hollywood Principle). Es liegt in der Verantwortung des Reactors (nicht der Anwendung), auf Events synchron zu warten und diese an den Event Handler zu delegieren und darauf die entsprechende Hook Methode aufzurufen.  **Lösung**  Es wird ein Eventloop erzeugt (single-threaded), der darauf wartet, bis Events eintreffen, und diese dann von Eventhandlern abarbeiten lässt. Der Thread bearbeitet immer gerade die Events von Ressourcen, die ready sind, also nicht blockieren.  **Struktur**  Handle  Ein Handle ist ein Identifikator auf eine Ressource, die Events generiert, z.B. ein Netzwerksocket oder ein File. Wenn die Ressource ein Event schickt, wird der Event im Handle in eine Liste gestellt und der Handle wird als “Ready” markiert.  Synchronous Event Demultiplexer  Eine Methode, die ein Set von Handles “Handle Set” überwacht. Sie blockiert solange, bis mindestens ein Handle “Ready” ist.  Sobald sie retourniert bedeutet das, dass auf einem Handle ein nicht blockierender Call ausgeführt werden kann.  Event Handler  Ein Interface, welches mehrere Hook Methoden anbietet. Diese Mehoden bieten mögliche Operationen zur Verarbeitung der applikations-spezifischen Events der Handles an.  Ein konkreter Event Handler gehört zu genau einem Handle. Er implementiert die Hook Methoden zur Verarbeitung der Events, welche durch ihren assoziierten Handle empfangen werden.  Reactor  Die Applikation registriert Event Handlers beim Reactor. Sobald der Event Loop des Reactors gestartet wird, sammelt dieser von all seinen registrierten Event Handlern die jeweiligen Handles ein und erstellt sich daraus sein Handle Set.  Mit dem Synchronous Event Demultiplexer wartet er nun, bis ein Handle aus dem Handle Set 'ready' wird. Dann dispatcht er den Event an den Event Handler.  Der Concrete Reactor implementiert das Reactor Interface, um so betriebssystem-spezifische Implementationen zu ermögliche.  **Ablauf**  https://wiki.ifs.hsr.ch/APF/files/reactor-workflow.png   1. Die Applikation registriert den konkreten Event-Handler beim Reactor 2. Der Reactor baut sich ein Set von registrierten Handles zusammen (eventHandler.getHandle()) 3. Die Applikation startet die Event-Loop beim Reactor (reactor.handleEvents()) 4. Der Reactor liest beim Syncronous Event Demultiplexer das erste Event aus. Ist noch kein Event verfügbar, blockiert der Demultiplexer. 5. Der Reactor bestimmt dann anhand des Events den entsprechenden konkreten Event Handler und ruft die entsprechende Hook Methode auf (Dispatch Event) 6. Der konkrete Event Handler ruft dann die anwendungsspezifische Methode auf, welche den Event verarbeitet.    1. Wenn die anwendungsspezifische Methode einen Rückgabewert hat, kann dieser in den Handle geschrieben und dem Client zurückgesendet werden. 7. Wiederholen ab Schritt 4. Der nächste Event wird verarbeitet   **Layering**  Das Pattern wird in zwei Layer unterteilt:  Demultiplexing/Dispatching Layer  Beinhaltet applikationsabhängig Funktionalität, wie dem Demultiplexieren und Aufrufen der Hook Methoden auf dem Event Handler (Dispatching)  Application Layer  Dieser Layer definiert konkrete Eventhandler, die in ihren Hook-Methoden die anwendungsspezifische Verarbeitung durchführen.  **Known Uses**  Telefonie  Die Teilnehmer im Telefonie-Netz sind Event Handler, welche sich beim Telefonie-Anbieter (Reactor) registrieren. Die Telefonnummer ist der Handle für jeden Teilnehmer. Wenn jemand diese Telefonnummer anruft, gibt es einen 'call-request' Event. Der Telefonie-Anbieter leitet den Anruf an den Teilnehmer weiter, welcher den Anruf verarbeitet.  **Abstract Window Toolkit**  Das Event Handling bei AWT ähnelt dem Reactor Pattern. Der Entwickler registriert Listener z.B bei einem Button und wird über Events (z.B MouseOver) benachrichtigt.  **Konsequenzen**  **Vorteile**   * Separation of concerns: Der Applikationscode muss sich nicht mit Demultiplexing beschäftigen. Anwendungsentwickler sind daher nur für die Implementierung von konkreten Eventhandlern verantwortlich und können die Demultiplex- und Dispatching-Mechanismen des Reaktors wiederverwenden * Reusability: Der Reactor ist application independent und kann wiederverwendet werden * Reusability: Durch den Eventmechanismus können die einzelnen Handler generischer implementiert werden. Z.B. einen Handler der Connections entgegennimmt. Dieser kann dann einfacher wiederverwendet werden. * Grobe Synchronisierung: Aufrufe werden schon beim Event synchronisiert. Der Applikationscode muss sich deshalb nicht mehr um Synchronisierung kümmern.   **Nachteile**   * Das Pattern skaliert nicht bei einer grosse Anzahl von parallelen Client-Anfragen und/oder Anfragen mit langer Laufzeit, da im Demultiplexing Layer, alle Events serialisiert werden. (Abhilfe: Proactor Pattern) * Ist nur dann effizient, wenn das Betriebssystem “Synchronous event demultiplexing” bereits unterstützt * Event Handler laufen im Thread des Reactors, sofern single-threaded. Das heisst sie dürfen keine Aufrufe machen, die blockieren oder sehr lange dauern. * Schwierig zu debuggen und zu testen   **Verwandte Patterns**   * Observer, Publisher - Subscriber * Chain of Responsibility: Im Gegensatz zum Reactor, der für jeden Event einen entsprechenden Event Handler bereit hält, durchsucht das Chain of Responsibility Pattern die Kette nach dem ersten passenden Event Handler. * Proactor: Der Reactor ist die synchrone Variante des asynchronen Proactors |
| **Asynchro-nouse Completion token** | Eine Applikation empfängt die Antwort asynchroner Operationen. Mit dem *Asynchronous Completion Token* Pattern ist es möglich, diese einheitlich zu serialisieren und zu verarbeiten.  **Alternative Begriffe** *Active Demultiplexing*, *‘Magic Cookie’*  **Kontext**  Event-getriebenes System, in welchem eine Applikation mehrere asynchrone Dienste in Anspruch nimmt.  **Problem**  Wenn eine Applikation (*client*) auf einen asynchronen *operation request* eine Antwort (*completion event*) erhält, muss sie diesen *demultiplex*ieren, also dem entsprechenden *Eventhandler* (einer Funktion oder einem Objekt) zuweisen, welcher die eigentliche Antwort verarbeitet.  Dabei gibt es drei Hauptprobleme:   * Der ursprüngliche Aufruferkontext ist dem Dienst nicht klar. So könnte der Aufrufer die Anfrage aus einem eigenen, zugehörigen Thread gestartet haben oder auch singlethreaded arbeiten. Der Service weiss somit nicht, welche Informationen der Client benötigt um den Response zu demultiplexieren. => d.h der Client sollte bestimmen, wie demultiplexiert wird. * Der Kommunikations-Overhead sollte so klein wie möglich sein, damit auch die Latenz klein bleibt. (Im besten Fall 1 Service Request und 1 Completion Event Response) * Das Objekt dass für das Demultiplexieren verwendet wird (ACT), sollte so wenig Speicherplatz wie möglich benötigen. * Die Applikation sollte für die Zuordnung vom *completion event* zum *handler* (Demultiplexing) so wenig Zeit wie möglich benötigen.   **Lösung**  Mit jeder asynchrononen Operation die ein Client *initiator* auf einem Service aufruft, wird ein Token mitgeschickt. Dieses Token bestimmt den zuständigen *completion handler*, welche die Antwort des Services entgegennimmt und verarbeitet. Das Token wird vom Service unverändert zusammen mit der Antwort an den *initiator* zurückgeschickt. Dort wird es benötigt, um die Antwort effizient zu demultiplexen und zu verarbeiten.  **Struktur**  https://wiki.ifs.hsr.ch/APF/files/act_model.png  Dienst (*service*)  Bietet beliebige Funktionalität, welche asynchron (z.B. über das Netzwerk) zugegriffen werden kann.  Applikation (client *initiator*)  Beansprucht den Dienst asynchron (*operation request*) und empfängt den *completion event*. Er ist für das Demultiplexing an den Completion Handler zuständig.  *completion handler*  Funktion oder Objekt, das die Antwort des Dienstes verarbeitet.  *asynchronous completion token (ACT)*  Identifikation eines *completion handler*s. Der Initiator sendet bei jedem *operation request* ein ACT mit. Der Service gibt dieses dann, sobald er fertig ist, mit der Response wieder zurück.   * Pointer ACT: Das ACT ist ein Pointer auf eine Funktion oder Objekt. Erfordert ähnliche Plattformen, wo ein Pointer gleich viele Bytes hat. Dieses Struktur wird oft eingesetzt. * Objectreference ACT: Das ACT ist ein Objekt. Erfordert eine Middleware, die das Objekt korrekt überträgt. Unterstützt unterschiedliche Plattformen * Index ACT: Index in eine Tabelle von CompletionHandlers. Ermöglicht persistente Handlers & unterstützt Sprachen, die keine Pointers haben.   Ablauf  https://wiki.ifs.hsr.ch/APF/files/act_process.png   1. Die Applikation erstellt ein *ACT* das den *completion handler* der Operation identifiziert. 2. Der *operation request* wird mit dem *ACT* an den *service* gesendet. 3. Die Applikation kann irgendwelche andere Tätigkeiten erledigen bis... 4. Der *service* sendet eine Rückmeldung, den *completion event*, welcher das *ACT* und die Antwort enthält. 5. Die Applikation verwenden das *ACT*, um den richtigen *completion handler* mit der Antwort als Parameter aufzurufen.     **Known Uses**  HTTP-Cookies  Web-Server kann ein Cookie mit einer ID (z.B. ein Session Token) beim GET setzten, wenn er eine POST-Antwort erwartet.  asynchrones IO  Wird bei Windows NT und POSIX für Asynchrone Operationen (wie Dateizugriffe) verwendet.  FedEx inventory tracking  Der Versender kann eine Bestellungs-ID angeben, welche nach dem Abgeben des Paketes beim Empfänger dem Versender mitgeteilt wird.  **Konsequenzen**  Vorteile   * **Vereinfachte Datenstrukturen** in der Applikation. Der Initiator kann die Responses auf eine einfache Art und Weise den entsprechenden Completion Handler weiterleiten. * Das Resultat im Response muss nicht geparst werden, um den korrekten Completion Handler zu bestimmen. * *ACT*s können leicht verarbeitet werden (z.B. wenn sie Pointer darstellen) * *ACT*s benötigen wenig Speicher (z.B Pointer ACT) * **Flexibel**: *ACT*s sind nicht an einen bestimmten Zweck gebunden. * **Offene Parallelität**: es werden keine spezielle Anforderungen an die Nebenläufigkeit gestellt. Der Initiator kann sowohl single- als auch multi-threaded sein.   Nachteile   * **Memory leaks / Application re-mapping**: wenn *ACT*s pointer repräsentieren, z.B. wenn die Applikation neu gestartet wird und im dynamic memory Adressen ändern. * **Nötige Authentisierung**: Wenn dem Server nicht vertraut werden kann, das ACT nicht zu verändern, muss jedes Response-ACT validiert werden. Der Server hätte es ja ändern können.   **Verwandte Patterns**   * Memento Patterns |
| **Proactor** | Das Proactor-Pattern erlaubt effizientes Demultiplexieren und Dispatchen von asynchronen Serviceanfragen. Das Pattern erreicht die Performance von nebenläufigen Programmen ohne die Konsequenzen von Paralleler Programmierung.  **Kontext**  Eine event getriebene Applikation die mehrere Service Anfragen asynchron empfängt und verarbeitet.  **Problem**  Beispiel  Ein Webserver verarbeitet HTTP Requests von mehreren Clients. Der Client verbindet mit dem Server (CONNECT), der Server öffnet die via GET angefragte Datei und sendet sie dem Client zurück. Mit dem uns bekannten Reactor Pattern würde bei jedem CONNECT ein Event Handler erstellt werden. Dieser Handler ist beim Reactor registriert, welcher für das synchrone Demultiplexieren und Dispatchen verantwortlich ist. Das Problem: Die Lösung skaliert nicht.   * Das Betriebssystem muss asynchrones I/O unterstützen. * Die Applikation sollte mehrere simultane Completion Events erlauben. Lang laufende Operationen sollten unterbunden werden. (Verbessert Skalierbarkeit und Latency) * Keine Synchronisation oder Kontext Switches sollten nötig sein (Maximiert Durchsatz) * Neue Services sollten mit minimalem Aufwand integriert werden können. * Im Applikationscode sollten keine Synchronisationsprimitiven nötig sein.   **Lösung**   * Unterteile den Service in zwei Komponenten: * Operationen   + Lang andauernde Operationen die asynchron ausgeführt werden * Completion Handler   + Handler die das Resultat der Operationen verarbeiten.   **Struktur**  Handle   * Identifizieren eine von Betriebsystem zur Verfügung gestellte Ressource wie z.B eine Netzwerkverbindung oder offene Datei. * Eine solche Ressource kann Completion Events generieren.   Initiator   * Ruft eine Asynchronous Operation auf * Verarbeitet oft auch das Resultat einer Asynchronous Operation die er aufruft. Er spielt dann auch die Rolle des Concrete Completion Handler.   Asynchronous Operation Processor   * Wird vom Betriebsystem Kernel implementiert. * Führt eine Asynchronous Operation aus. * Wenn die Asynchronous Operation terminiert, fügt der Processor einen Completion Event in die Completion Event Queue ein.   Asynchronous Operation   * Repräsentiert eine potentiell langlaufende, asynchrone Operation * Für jeden Service der von einer Applikation angeboten wird, gibt es eine Asynchronous Operation. * Beim Ausführen wird der Aufrufer-Thread nicht blockiert (asynchronous)   Completion Event   * Enthält das Resultat einer Asynchronous Operation   Completion Event Queue   * Buffer von Completion Event * Wird vom Proactor abgearbeitet   Asynchroner Event Demultiplexer   * Eine Funktion, die darauf wartet, dass Completion Events in die Completion Event Queue eingefügt werden. * Entfernt den Completion Event und gibt ihn dem Aufrufer zurück. (Proactor) * Blockiert wenn die Completion Event Queue leer ist.   Proactor   * Kontrolliert Event Loop. Die Event Loop wird vom Initiator ausgeführt. * Fordert den Asynchroner Event Demultiplexer auf, einen Completion Event zu entfernen. * Demultiplexiert und Dispatched den Completion Event zum applikationsspezifischen Completion Handler.   (concrete) Completion Handler   * Implementiert Hook Methoden, die das Resultat der Completion Events verarbeiten. * Ist assoziiert mit einem Handle, worüber er wiederum selber eine Asynchronous Operation aufrufen kann. (z.B Verarbeitet ein Read und ruft dann die Write Operation auf)   https://wiki.ifs.hsr.ch/APF/files/proactor-structure.png  **Ablauf**  https://wiki.ifs.hsr.ch/APF/files/proacter_ablauf.png   1. Der Initiator ruft eine Asynchronous Operation auf einem Asynchronous Operation Processor auf. (exec\_asyc\_operation())    1. Er übergibt Parameter zur Identifikation des Completion Handler oder ein Handle zur Completion Handler Event Queue.    2. Der Initiator kann sich auch selber als Completion Handler übergeben.    3. Diese Parameter werden vom Asynchronous Operation Processor intern gespeichert. 2. Der Asynchronous Operation Processor ruft die Asynchronous Operation auf. (async\_operation()) 3. Nun läuft der Initiator und die Asynchronous Operation unabhängig voneinander. 4. Sobald die Asynchronous Operation terminiert, wird vom Asynchronous Operation Processor ein Completion Event erstellt.    1. Das Completion Event enthält das Resultat des Services 5. Der Asynchronous Operation Processor fügt den Completion Event in die Completion Event Queue ein. 6. Sobald die Applikation (Initiator) die Resultate verarbeiten möchte, startet sie die Event Loop auf dem Proactor. (handle\_events()) 7. Die Event Loop startet einen Asynchronous Event Demultiplexer, welcher auf Completion Events in der Completion Event Queue wartet. (blockierend) 8. Nachdem der Asynchronous Event Demultiplexer einen Event von der Queue entfernt hat, wird der Completion Event vom Proactor an den entsprechenden Concrete Completion Handler demultiplexiert. (handle\_event())    1. Das Resulat wird dem Completion Handler übergeben. 9. Sobald der Completion Handler das verarbeitete Resultat an seinen Aufrufer zurückgeben möchte, gibt es zwei Möglichkeiten    1. Initiator = Completion Handler: Das Resultat muss nicht an den Aufrufer zurückgegeben werden, da der Aufrufer bereits der Completion Handler ist.    2. Initiator != Completion Handler: Der Completion Handler ruft eine Asynchronous Operation (WRITE) auf. Über das Handle wird dann der korrekte Initiator benachrichtigt. 10. Die Event Queue läuft weiter: Einerseits werden neue Operation ausgeführt, sowie parallel die Resultate der Operationen verarbeitet.   **Known Uses**  Festnetz Telefonie (Real Life Example)  Eine Person (Initiator) ruft einen Freund an, der momentan nicht zu Hause ist. Er hinterlässt eine Nachricht (Asynchronous Operation) auf dessen Voice Mail (Asynchronous Operation Processor) und bittet ihn zurückzurufen. Während die Person auf den Rückruf wartet, kann sie beliebige Dinge erledigen. Sobald der Freund die Voice Mail abhört (Completion Event), ruft der Freund die Person zurück. Der Freund ist in diesem Fall der Proactor. Während dem Gespräch ist der Initiator der Completion Handler der den Rückruf verarbeitet.  **Konsequenzen**  Vorteile   * **Separation of concerns**: Applikationslogik und asynchrone Mechanismen sind strikt voneinander getrennt. Die applikationsunabhängigen Mechanismen können somit gut wiederverwendet werden. * **Portability**: Dank Interfaces ist nur das Demultiplexing Plattform abhängig. * **Encapsulation of concurrency mechanisms**: Da die Concurrency-Mechanismen im Proactor sind, können Proactors mit verschiedenen Concurrency Mechanismen konfiguriert werden (Dispatching in einem oder mehreren Threads) * **Decoupling of threading from concurrency**: Durch die asynchronen Operationen ist es nicht notwendig, viele Threads für die Verarbeitung zu starten. * **Performance**: Beim Proactor Pattern gibt es keine Context-Switches, was die Performance steigert. * **Simplification of application synchronization**: Das Pattern braucht wenig bis keine Synchronisationsmechanismen   Nachteile   * **Restricted applicability**: Das Betriebsystem muss asynchronous I/O nativ unterstützen. * **Complexity of programming, debugging and testing**: Durch die asynchrone Verarbeitung erhöht sich die Komplexität. Das Dispatching der Ereignisse ist nicht deterministisch und erschwert das Debugging. * **Scheduling, controlling, and canceling asynchronously running opera- tions**: Die Ausführungsreihenfolge ist schwer beeinflussbar.   **Verwandte Patterns**   * Reactor   vgl Woche 4. Proactor ist eine Art asynchroner Reactor   * Observer, Publisher-Subscriber   Im Observer Pattern geht es meist um eine einzige Quelle an Daten. Der Proactor dient jedoch dazu, Daten aus mehrere Quellen an die entsprechenden completion handlers zu verteilen. |
| **Acceptor Connector** | Entkoppelt die Verbindung und Initialisierung von kooperierenden Services in einem Netzwerk von der Verarbeitung, welche von diesen Services durchgeführt wird.  **Kontext**  Ein Netzwerk in welchem verbindungsorientierte Kommunikationsprotokolle genutzt werden, damit Services über ihre *Transport-Endpoints* kommunizieren können.  **Problem**  Applikationen in verbindungsorientierten Netzwerken besitzen oft eine Menge Konfigurations Code, um Verbindungen aufzubauen und Services zu initialisieren. Dieser Code hat meist keinen Zusammenhang zum Code, welcher die empfangenen Daten verarbeitet. Es macht also keinen Sinn, diese beiden Code-Bereiche eng zu koppeln.   * Es sollte einfach sein, Verbindungsrollen (aktives Aufbauen oder passives Entgegennehmen von Verbindungen) von Services zu ändern. * Es sollte einfach sein, neue Arten von Services, Service-Implementationen und Kommunikationsprotokolle einzuführen, ohne dass dies bestehenden Code zum Verbindungsaufbau und Service Initialisierung beeinflusst. (z.B FTP, Telnet, HTTP können alle TCP für den Verbindungsaufbau verwenden) * Grosse Netzwerke sollen von OS-Features wie asynchronen Verbindungs-Mechanismen profitieren können. (Reduziert Latenz beim Verbindungsaufbau)   **Lösung**  Trennung von:   * Code für den Verbindungsaufbau und die Initialisierung von Services * Code für die Verarbeitung von empfangenen Daten (effektiv angebotene Funktionalität eines Services)   Grundlegender Ablauf:   1. Ein Connector erstellt eine Verbindung zu einem Remote Acceptor und initialisiert einen Service Handler, welcher die Daten verarbeitet, sobald die Verbindung aufgebaut ist. 2. Der Acceptor wartet passiv auf Verbindungsanfragen von einem Remote Connector. Auch er erstellt einen Service Handler der für die Daten Verarbeitung zuständig ist. 3. Die initilisierten Service Handler sind für die applikationspezifische Verarbeitung zuständig. Sie kommunizieren über die Verbindung die von Connector und Acceptor erstellt wurden.   Etwas genauer:   * Jeder Service wird in einem *service handler* gekapselt, welcher einen Verbindungs-Endpunkt implementiert. * Diese handlers werden mit Hilfe von zwei Factories erstellt, welche zusammen jeweils eine vollständige Verbindung zwischen zwei Service-Endpunkten generieren.   *acceptor*: erstellt passive Verbindungen für einen entsprechenden *service handler* beim Eintreffen von *connection request events* von remote *service handler*.  *connector*: erstellt aktive Verbindungen für einen entsprechenden *service handler* zu einem anderen remote *service handler*   * Sobald die Verbindung aufgebaut ist, übergeben die acceptor und connector factories den entsprechenden transport handle zum peer service handler. Dieser ist für die applikationsabhängige Verarbeitung zuständig und nutzt die aufgebaute Verbindung für seine Kommunikation.   **Struktur**  https://wiki.ifs.hsr.ch/APF/files/acceptor-connector-hs17.png  Transport Endpoint   * Factory: erstellt transport handle * Akzeptiert eintreffende connection requests und kapselt den neu verbundenen transport endpoint in einem *transport handle*   Transport Handle   * Kapselt einen transport endpoint * Zwei transport endpoints übertragen Daten über die jeweiligen transport handles.   Service Handler   * Ist für die applikationsspezifische Verarbeitung von Daten zuständig. * spielt oft entweder die Rolle des Clients oder des Servers (oder beides in P2P setup) * Bietet activation hook Methode, um den handler nach dem Verbindungsaufbau zu initialisieren. * Wird von einem connector rsp. acceptor aktiviert. * Beinhaltet einen *transport handle*, welcher den Verbindungs-Endpunkt des handlers kapselt   Acceptor   * Factory: implementiert Strategie zum passiven Verbindungsaufbau * Bietet zwei Methoden   **connection initialization**: Bindet seinen *transport endpoint* an eine *transport address* (IP-Adresse:TCP-Port) und hört passive auf eingehende *connection requests*  **connection completion**:   * + 1. erstellt verbundenen *transport endpoint* und kapselt ihn in einem *transport handle*     2. erstellt *service handler*, welche empfangene Daten verarbeitet     3. speichert *transport handle* im entsprechenden *service handler* und ruft dessen *activation hook methode* auf, damit der *service handler* sich selber fertig initialisieren kann.   Connector   * Factory: implementiert Strategie zum aktiven Verbindungsaufbau * Durch die Unterteilung in die zwei folgenden Methoden unterstützt der *connector* synchronen sowie asynchronen Verbindungsaufbau.   connection initialization:   * + 1. Bekommt einen existierenden *service handler* und erstellt dafür einen entsprechenden *service endpoint* mit einem *acceptor*.   **connection completion**: Wird aufgerufen sobald die Verbindung akzeptiert wird.  Dispatcher   * Zuständig für die Demultiplexierung von Events wie connection requests und data requests * Registriert und demultiplext *acceptors*, *connectors* und *service handlers*.   **Ablauf**  Passiver Verbindungsaufbau   1. Phase: Transport Endpoint Initialisierung:    1. Acceptor erstellt *transport endpoint*    2. Acceptor registriert sich selber bei einem Dispatcher, welcher den Acceptor notifiziert wenn es *connection indication events* gibt.    3. Applikation startet die Dispatcher Event Loop 2. Phase: Service Handler Initialisiserung    1. Wenn ein Service Request eintrifft, wird der Acceptor vom Dispatcher notifziert    2. Der Acceptor erstellt neuen Transport Endpoint und kapselt ihn in einem Handle.    3. Dieses Handle wird dann am Service Handler übergeben und die *activation method* aufgerufen    4. Der Service Handler registriert sich beim Dispatcher, welcher eingehende Request direkt and den Service Handler weiterleitet. 3. Phase: Service Processing    1. Sobald die Verbindung aufgebaut ist, kann mit einem Layer 7 (application) Protokoll Daten zwischen den Endpoints ausgetauscht werden.   https://wiki.ifs.hsr.ch/APF/files/acceptor.png  Aktiver synchroner Verbindungsaufbau   * Synchroner Verbindungsaufbau ist dann nützlich wenn:   + Verbindungsaufbau nur sehr kurz dauert (z.B Loopback Interface)   + Man ein Thread-per-Connection Model einsetzt   + Die Reihenfolge des Verbindungsaufbau fix ist.  1. Phase: Verbindung Initialisierung    1. Connector wird von der Applikation initialisiert    2. Connector erstellt eine synchrone Verbindung    3. Der Applikation Thread wird dabei blockiert 2. Phase: Service Handler Initialisierung    1. Der Connector initiiert die Initialisierung des Service Handler    2. Der Applikations Thread blockiert immer noch 3. Phase: Service Processing    1. Die Verarbeitung der Daten ist ähnlich wie beim Acceptor Setup.   https://wiki.ifs.hsr.ch/APF/files/connector_sync.png  Aktiver asynchroner Verbindungsaufbau   1. Phase: Verbindung Initialisierung    1. Die Applikation gibt dem Connector einen Service-Handler zur Verbindung und ruft "connection initiation" auf    2. Der Connector nutzt den Transport-Handle aus dem Service-Handler und schickt einen Connection-Request an den entfernten Service. 2. Phase: Service-Handler Initialisierung    1. Der Connector trägt sich beim Dispatcher für den Connection event ein    2. Der Dispatcher ruft "connection completion" auf dem Connector auf    3. Der Connector ruft die Hook-Methode auf dem Service-Handler auf 3. Phase: Service processing    1. Der Service-Handler trägt sich beim Dispatcher für eingehende Request-Events ein    2. Der Dispatcher notifiziert den Service-Handler bei eingehendem Request    3. Der Service-Handler verarbeitet den Request   https://wiki.ifs.hsr.ch/APF/files/connector_async.png  **Known Uses**  WebBrowser: Die HTML Parser von Webbrowser verwenden asynchrone Connectors  CORBA Object-Request Broker verwendet dieses Pattern  **Konsequenzen**  Vorteile   * Wiederverwendbarkeit, Portabilität und Erweiterbarkeit: Die anwendungsunabhängige Logik für die Initialisierung und den Verbindungsaufbau ist komplett getrennt von der Anwendungslogik. (separation of concerns) * Robustheit: Durch die Trennung zwischen Service Handler und Acceptor wird sichergestellt, dass nicht fälschlicherweise das passive Endpoint-Handle zum Schreiben und Lesen von Daten verwendet wird. * Effizienz: Durch die asynchrone Strategie können viele Verbindungen gleichzeitig geöffnet werden.   Nachteile   * Zusätzliche Indirektion im vergleich zur direkten Verwendung der Netzwerkschnittstelle. * Erhöhte Komplexität: Für Anwendungen die nur mit einem Server sprechen ist das Pattern unnötig kompliziert.   **Verwandte Patterns**  Client-Dispatcher-Server (POSA1): Beide sind für die Trennung von Verbindungsaufbau und Verarbeitung nützlich. Der Unterschied ist, dass das Client-Dispatcher-Server Pattern nur synchronen Verbindungsaufbau unterstützt. |
| **Compo-nent Configura-tor** | Einbinden, konfigurieren und entfernen von Komponenten zur Laufzeit ohne Code-Änderungen (sprich ohne: Source Code ändern, Neu Kompilieren, Neu Linken). Darüber hinaus unterstützt der Component Configurator das Einfügen von Komponenten in verschiedene Anwendungsprozesse, ohne dass dieser Prozess heruntergefahren und neu gestartet werden muss.  **Alternative Begriffe** Service Configurator  **Kontext**  Eine Anwendung oder ein System, in der Komponenten so flexibel und transparent wie möglich initiiert, pausiert, fortgesetzt und terminiert werden müssen.  **Problem**  Applikationen, die aus mehreren Komponenten bestehen, benötigen einen Mechanismus, um die Komponenten zu konfigurieren.  Dazu müssen die folgenden Bedingungen erfüllt werden:   * Die Implementation einer Komponente sollte zur Laufzeit ausgetauscht / neugestartet werden können, ohne andere Komponenten gross zu behindern * Es soll spät im Entwicklungsprozess noch möglich sein, die Konfiguration der Komponenten zu ändern * Administrationsaufgaben wie Konfigurieren/Initialisieren/Kontrollieren von Komponenten soll so einfach wie möglich sein. Am besten geht das oft mit einer zentralen Instanz.   **Lösung**  Alles wird gegen Interfaces programmiert. Die *Concrete Component*s implementieren dieses. Ein *Component Configurator* kann zur Laufzeit die *Concrete Component*s austauschen.  **Struktur**  https://wiki.ifs.hsr.ch/APF/files/component-configurator-class-diagram.png  Component:  Einheitliches Interface für die Implementation einer Komponente. Bietet häufig Methoden an für: Initialisierung/Pausieren/Weitermachen/Terminieren.  *Zustände / Methoden eines Components*  https://wiki.ifs.hsr.ch/APF/files/component-configurator-state-diagram.png  Concrete Component:  Implementiert das *Component*-Interface. Die konkrete Komponente wird paketiert z.B. in eine DLL. (Dynamic Link Library)  Component Repository:  Ein Repository, das alle *Concrete Component*s verwaltet, die momentan im System aktiv sind. Beim Repository können neue Komponenten eingetragen und alte wieder ausgehängt werden.  Component Configurator:  Der Component-Configurator enthält ein *Component Repository*. Er verwendet dieses zur (Re-)Konfiguration von Komponenten im System.  **Ablauf**  Die Pattern Dynamics werden in drei Phasen eingeteilt:   1. **Component initialization** durch *Component Configurator*    1. Laden und initialisieren der *Concrete Component*: *init()*    2. Hinzufügen der *Concrete Component* zum *Component Repository*: *insert()* 2. **Component processing** durch Applikation und *Concrete Component*    1. Ausführen der eigentlichen Aufgabe der *(Concrete) Component*: *run\_component()*    2. Die Komponente kann durch den *Component Configurator* temporär suspendiert, und wieder aktiviert werden: *suspend()*, *resume()* 3. **Component termination** durch *Component Configurator*    1. Beenden nicht mehr benötigte *Concrete Component*: *fini()*    2. Entfernen der Komponente aus dem *Component Repository*: *remove()*   Beispiel Sequenzdiagramm  https://wiki.ifs.hsr.ch/APF/files/component-configurator-sequence-diagram.png  **Known Uses**  Gerätetreiber  Moderne Betriebssysteme wie Windows oder Linux implementieren Treiber mit einer Form des Component Configurators.  Java Applets  Verwaltung von Java Applets im Browser, welche auf Webseiten eingebunden sind.  Fussball  Spieler können während des Spiels ausgetauscht werden. Die Liste mit den aktuellen ≤11 Spielern entspricht dem *Component Repository*. Der Coach (*Component Configurator*) entscheidet welche Spieler aktuell spielen dürfen.  **Konsequenzen**  Vorteile   * Einheitlichkeit: Alle Komponenten haben dasselbe Interface * **Zentralisierte Administration**: Dieses Pattern gruppiert eine oder mehrere Komponenten in eine administrative Einheit. Tasks können so besser automatisiert werden * **Modularität**: Da alle Komponenten über ein einheitliches Interface verfügen, lassen sich monolithische Anwendungen leichter in wiederverwendbare Komponenten zerlegen, die unabhängig voneinander entwickelt und getestet werden können. * **Reusability**: Die einzelnen Komponenten sind losgelöst von Anderen. Sie können einfacher wiederverwendet werden. * **Dynamische Konfiguration/Tuning**: Komponenten können zur Laufzeit den Anforderungen entsprechend konfiguriert und angepasst werden * **Testbarkeit**: Einzelne Komponente können isoliert getestet werden.   Nachteile:   * **Nichtdeterminismus**: Je nach Konfiguration verhalten sich Komponenten jeweils anders und unvorhersehbar. Das Laufzeitverhalten einer Applikation ist schwer vorherzusehen, da Komponenten zur Laufzeit hinzu konfiguriert werden können. * **Weniger stabil** als statisch konfigurierte Applikationen. * **Weniger sicher**, weil böswillige Komponenten eingeschleust werden könnten. * **Runtimeoverhead / Komplexität**: Jede Komponente muss immer bei der Registry ein/ausgetragen werden, was die Performance beeinträchtigt. * **Zu unspezifische Interfaces**: Die Initialisierung einer Komponente kann je nach Komplexität mehr Methoden benötigen als die einfachen init/finish Methoden gemäss *Component*-Interface.   **Verwandte Patterns**  Das *Configuration Pattern* entkoppelt in ähnlicher Weise die Initialisierung der Komponenten von der Komponentenverarbeitung. Der Hauptunterschied besteht darin, dass sich das Configuration Pattern auf die aktive Zusammensetzung von Ketten verwandter Protokolle und Dienste konzentriert. Im Gegensatz dazu fokusiert sich das *Component Configurator Pattern* auf die dynamische Initialisierung von Komponenten, welche Requests die zwischen Transport Endpunkten ausgetauscht werden, verarbeitet. |
| **Active Object** | Entkoppeln der Methodenausführung vom Methodenaufruf, um die Parallelität zu verbessern und Synchronisation beim Zugriff auf Objekte in eigenen Threads zu vereinfachen.  **Alternativer Begriff** Concurrent Object  **Kontext** Applikationen, welche auf Objekte in separaten Threads zugreifen.  **Problem**  Wenn eine Applikation parallel auf Objekte zugreift, muss der Zugriff von extern auf Methoden und Daten synchronisiert werden.  Dabei gibt es folgende Problemstellungen:   * Rechenintensive Aufrufe sollten den aufrufenden Prozess nicht endlos blockieren. * Die Synchronisation sollte Methodenaufrufe nicht komplizierter machen. * Das Serialisieren und Scheduling für Zugriffe auf synchronisierte Objekte sollte für den Aufrufer unsichtbar passieren. * Die verfügbaren Ressourcen (Harware) sollten gut genutzt werden. (Multicore CPU)   **Lösung**   * Trennen von Methodenaufruf und Ausführung. * Methodenaufruf sollte im Aufrufer-Thread (*Client*-Thread) passieren. * Die Methodenausführung sollte in einem eigenen Thread ablaufen. * Der Aufrufer (*Client*) sollte davon nichts bemerken   Eine *Proxy* (POSA1/[GoF](https://wiki.ifs.hsr.ch/APF/wiki.cgi?GoF&printerfriendly)) repräsentiert das Objekt mit der aufzurufenden Methode, welches von einem *Servant* tatsächlich implementiert wird. Der *Proxy* läuft im Thread des Methodenaufrufers (*Client*) und delegiert Zugriffe auf den *Servant* in einen eigenen Thread. Zudem liefert der *Proxy* dem *Client* ein *Future* zurück, in welchem die Rückgabewerte der aufgerufenen Methode gespeichert werden.  Zwischen dem *Proxy* und *Servant* läuft zudem ein *Scheduler*. Dieser serialisiert Methodenaufrufe mittels einer Warteliste und einer Event-Loop, in welcher die eigentlichen *method request*s ausgeführt werden.  **Struktur**  https://wiki.ifs.hsr.ch/APF/files/active-object-domain-model.png  *Client*: Startet Methodenaufruf über den *Proxy* und erhält ein *Future* zurück  *Proxy*:   * Repräsentiert das Interface des *Active Object* gegenüber dem *Client* * Das Interface enthält die öffentlich verfügbaren Methoden des *Active Object*s * Läuft im Client Thread * Erzeugt ein Method Request Object   *Servant*:   * Implementation des *Active Object* * Läuft in einem eigenen *Active Object* Thread   *Scheduler*:   * Kontrolliert die Event Loop die im gleichen Thread wie der *Servant* läuft * Entfernt Methodenaufrufe von der *Activation List* und dispatched sie an den *Servant*   *Activation List*:   * Enthält *Method Request* die ausgeführt werden sollen * Wird vom Proxy befüllt * Wird vom Scheduler entleert   *Future*:   * Enthält des Resultat eines Methodenaufrufes auf einem *Active Object* * Wird vom Proxy zurückgegeben   *Method Request*:   * Repräsentiert einen Methodenaufruf auf dem *Active Object* * Enthält Kontext Informationen wie Methoden Parameter * Bietet eine *guard()* Methode zum überprüfen, ob der *method request* ausgeführt werden kann. * Definiert ein Interface für die Ausführung eines *Active Object* * Für jede public Methode des Proxys, welche Synchronisation benötigt, wird eine konkrete Implementierung der Method Request Klasse erstellt.   *Concrete Method Request*:   * Für jede öffentliche Methode die auf dem Proxy definiert ist, gibt es eine *Concrete Method Request* Klasse. * Repräsentiert einen spezifischen Methodenaufruf   **Ablauf**  https://wiki.ifs.hsr.ch/APF/files/active-object-workflow.png  Drei Phasen:   1. **Method request construction and scheduling**:    1. Der *Client* ruft eine Methode auf dem *Proxy* auf    2. Der *Proxy* erstellt ein *Method Request* Objekt. Hat der *Method Request* einen Rückgabewert wird bei der Erstellung ein *Future* Objekt mitgegeben.    3. Der *Proxy* übergibt das *Method Request* Objekt dem *Scheduler*    4. Der *Scheduler* trägt das neue Objekt in der *Activation List* ein    5. Wenn die Methode einen Rückgabewert hat (Two-Way Invocation) dann gibt der *Proxy* dem *Client* ein *Future* zurück. In diesem kann später das Resultat ausgelesen werden. 2. **Method request execution**:    1. Der *Scheduler* läuft in einem anderen Thread als Client    2. Der *Scheduler* ruft die *guard()* Methode auf dem *Method Request* Objekt auf und prüft damit, ob die Methode ausführbar ist.    3. Wenn eine Methode ausführbar ist, wird sie durch den *Scheduler* von der *Activation List* entfernt. Auf dem *Servant* wird die entsprechende Methode aufgerufen. 3. **Completion**:    1. Sobald ein Resultat vorhanden ist, wird es im *Future* abgespeichert    2. *Client*s können das Resultat im *Future* lesen (Hier benötigt es **Synchronisation**!!)   **Known Uses**  Java  bietet Timer-based tasks, mit java.util.Timer und java.util.TimerTask, welche Tasks nach einem Timeout in einem eigenen Thread ausführt.  Restaurant-Koch  Die Bedienung gibt asynchron Essensbestellungen beim Koch auf, welcher die Aufträge in einer Liste effizient verwaltet und ordnet. Nach der Zubereitung wird der Bedienung das Essen mit der ursprünglichen Bestellung zurückgegeben.  **Konsequenzen**  Vorteile   * **Vereinfacht Parallelität und Synchronisation**: Client und Methodenaufrufe sind asynchron in eigenen Threads. Der *Scheduler* garantiert den serialisierten Zugriff. * **Transparente Ausnutzung von Parallelität**: Erlaubt eine flexible Ausnutzung der Parallelität je nach System/Hardware * **Unabhängige Methodenaufrufe**: Methodenaufrufe sind nicht voneinander abhängig. * **Flexibilität**: Die Reihenfolge in der Methoden aufgerufen wurde kann sich unterscheiden zu der Reihenfolge in der die Methode effektiv ausgeführt werden. (je nach guard). Dies verbessert die Performance und Flexibilität.   Nachteile   * **Performance overhead**: Durch den *Scheduler* kann je nach Implementation beim Kontext-Switching, Synchronisieren und Data-Movement viel Zeit verloren gehen. * **Schwieriges Debugging**: Durch die Parallelität und das nicht deterministische Verhalten wird die Fehlersuche stark erschwert.   **Verwandte Patterns**   * **MonitorObject** Hauptunterschied: Die *activation list* des **Active Object** Patterns unterstützt komplexe scheduling Strategien. Die asynchrone Ausführung von mehreren Operationen gleichzeitig wird unterstützt. Dafür ist das Implementieren der *Message Queue* im **Monitor Object** Pattern einfacher und oft auch performanter. * **Reactor** wird oft an Stelle des **Active Object**s verwendet, um Callback Operationen an passive Objekte zu planen.  Das **Reactor** Pattern kann auch zusammen mit dem **Active Object** Pattern verwendet werden, um das **HalfSyncHalfAsync** Pattern zu bilden. * C**ommand Processor Pattern** (POSA1) : Unterstützt Parallelität **nicht** -> alles läuft im selben Thread ; Es gibt keine *Proxies*, stattdessen werden *Commands* direkt an den *Command Processer* übergeben. * Broker Pattern (POSA1) : Im **Broker** sind *Servants* und *Proxies* durch die Verteilung im Netzwerk getrennt. Beim **Active Object** sind sie durch die Verteilung auf verschiedene Threads getrennt. **Active Object** hat typischerweise nur einen *Servant*, wohingegen das **Broker** Pattern mehrere *Servants* haben kann. |
| **Monitor Object** | Das *Monitor Object* Design Pattern synchronisiert parallele Methodenaufrufe auf ein Objekt, damit nur eine Methode gleichzeitig ausgeführt wird.  **Alternative Begriffe**  Thread-safe Passive Object. Passive Objekte sind Objekte, die den Thread ihres Aufrufers benutzen, um eine Methode auszuführen.  **Kontext**  Mehrere Threads, welche gleichzeitig auf ein Objekt zugreifen möchten.  **Problem**  Sobald beim parallelen Aufruf eines Objekts Zustandsänderungen gemacht werden können (in erster Linie Instanzvariablen oder Seiteneffekte), muss der Zugriff serialisiert werden.  Dabei gibt es folgende Problemstellungen:   * Die Methoden des Objektinterfaces sollten ihre Synchronisations-Eigenschaften definieren * Es darf jeweils nur eine Methode eines Objekts ausgeführt werden * Das verwenden von Synchronisationsprimitiven wie Mutexes, Semaphoren , Condition variablen ist kompliziert. * Nebenläufige Anwendungen sind schwieriger zu programmieren, wenn dafür Low Level Synchronisationsprimitiven wie Semaphoren, Mutexe oder Condition Variables verwendet werden.   Objekte sollten daher selber dafür verantwortlich sein, dass alle ihre Methoden, die synchronisiert werden müssen, transparent serialisiert werden, ohne dass der Client dies explizit fordert.   * Wenn eine Methode im Objekt blockiert, sollte das Objekte während dieser Zeit für andere freigegeben werden. * Nach jedem Zugriff sollte das Objekt einen gültigen Zustand haben.   **Lösung**  Synchronisieren der Methodenaufrufe, damit nicht mehrere gleichzeitig ausgeführt werden können.  Konkret:   * Für jedes Objekt, auf das mehrere Client-Threads gleichzeitig zugreifen, wird ein Monitorobjekt definiert. * Die Methoden davon können nur durch *Synchronized Methods* aufgerufen werden. * Jedes *Monitor Object* enthält einen *Monitor Lock* * Der *Monitor Lock* stellt sicher, dass nur jeweils eine synchronisierte Methode läuft. * Synchronisierte Methoden können mittels *Monitor Conditions* spezifizieren, wie sie *suspended* und *resumed* werden.   **Struktur**  https://wiki.ifs.hsr.ch/APF/files/monitor-object-domain.png  Monitor Object:   * Definiert das Objekt, auf welches nebenläufig zugegriffen wird * Alle Clients müssen die Methoden des Monitor Objekts verwenden   Synchronized Method: Wird im Thread des Aufrufers ausgeführt  Monitor Lock:   * Jedes Monitor Objekt hält ein Monitor Lock * Jede Synchronized Method muss das Lock beziehen, bevor sie den Monitor betritt.   Monitor Condition: Mehrere synchronisierte Methoden, die in getrennten Threads laufen, können ihre Ausführungssequenzen kooperativ planen, indem sie auf einander warten und sich gegenseitig über Monitor conditions benachrichtigen, die mit ihrem Monitorobjekt verknüpft sind.  **Ablauf**  https://wiki.ifs.hsr.ch/APF/files/monitor-object-sequence.png   1. Synchronized method invocation and serialization    * Wenn Client Thread 1 eine synchronisierte Methode eines *Monitor Objekts* aufruft, muss das Objekt als erstes den *Monitor Lock* erhalten. Dies geht nur, wenn kein anderer Thread bereits eine Methode des Objekts ausführt. 2. Synchronized method thread suspension    * Wenn eine synchronisierte Methode nicht ausgeführt werden kann, kann auf die *monitor condition* gewartet werden (*wait()*). In diesem Fall wird der Client Thread suspendiert und der *Monitor Lock* wird freigegeben. (-> Das Monitor Object muss beim Verlassen in einem *stable state* sein!) 3. Monitor condition notification    * Eine synchronisierte Methode kann eine *monitor condition* benachrichtigen (*notify()*). Dadurch wird ein Client Thread, der auf diese *condition* wartet aufgeweckt. Ebenfalls können direkt alle Client Threads, die auf eine Condition warten aufgeweckt werden (*noritfyAll()*). 4. Synchronized method thread resumption   Sobald ein suspendierter Methoden-Thread aufgeweckt wird, kann er seine Ausführung an der Stelle fortsetzen werden, an der er auf die Monitorbedingung gewartet hat. Das Monitor Lock wird vom Thread Scheduler automatisch wieder bezogen.  **Known Uses**  Dijkstra and Hoare-style Monitors, z.B. in Java  Java erlaubt es, Methodenaufrufe mittels dem *synchronized* Schlüsselwort in synchronisierte *Monitor Object* Methoden umzuwandeln. *wait()* *notify()* und *notifyAll()* erlauben die Steuerung des Monitors bzw. *Monitor conditions*.  Fast-Food Restaurant  vor der Kasse bildet sich eine Warteschlange; der Kassier nimmt eine Bestellung nach der anderen auf. Sofern eine Bestellung nicht sofort abgehandelt werden kann, wartet der Kunde kurzzeitig neben der Warteschlange.  **Konsequenzen**  Vorteile   * Simplicty: Es ist für die Clients total transparent und einfach zu benutzen. Es ist auch relativ einfach zu implementieren * Vereinfacht Scheduling: Durch Wait&Notify können Threads einfach pausiert werden ohne aufwändiges Polling   Nachteile   * Scalability: Da es nur ein einziges Monitorlock gibt / geben darf, leidet die Performance zunehmend, mit der Anzahl Clients, die gleichzeitig zugreifen * Kopplung: Synchronisations und Scheduling Logik (Wait&Notify) ist direkt in der Methode. Diese Kopplung macht das Monitor zwar effizienter wie z.B das Active Object, dafür ist es schwieriger die Synchronisationslogik zu ändern ohne die Methoden Implementation des Monitor Objekt anzupassen. * Es kann nicht einfach & transparent von MonitorObjects abgeleitet werden. * Nested Monitor Lockout: Wenn MonitorObjects selber andere MonitorObjects halten, kann es zu Deadlocks kommen   **Verwandte Patterns**   * Code Locking pattern: Eine nicht-Objekteorientierte Variante dieses Patterns. * [U11\_2\_ActiveObject\_HS17](https://wiki.ifs.hsr.ch/APF/wiki.cgi?U11_2_ActiveObject_HS17&printerfriendly): Ähnliches Pattern, welches aber nicht in erster Linie die Zugriffe synchronisieren soll, sondern asynchrone Aufrufe kapselt. Das Active Object Pattern erlaubt allerdings erweiterte Scheduling-Strategien. Ebenfalls führt es die Methoden in einem eigenen Thread aus. |
| **Scoped Locking** | Das **Scoped Locking** stellt sicher, dass ein Lock automatisch erhalten und frei gegeben wird, wenn ein bestimmter *scope* im Programm durchlaufen wird. Auch bekannt als: *Synchronized Block*, *Guard*  **Kontext**  Eine Applikation enthält Ressourcen, welche von mehreren Threads nebenläufig bearbeitet werden.  **Problem**  Code in einer multi-thread Applikation, welcher nicht nebenläufig ausgeführt werden soll, muss durch einen Locking Mechanismus geschützt werden. Dabei muss ein Lock unbedingt bei jeglicher 'Art' des Verlassens (return, break, continue, goto, exception [handeled und unhandeled]) eines kritischen Abschnitts freigegeben werden. Bei manuellem Holen und Freigeben des Locks kommt es schnell zu Fehlern!  **Lösung**  Implementierung einer *Guard* Klasse, welche im Konstruktor automatisch ein Lock beansprucht und das Lock im Destructor automatisch wieder frei gibt.  **Struktur / Implementierung**  https://wiki.ifs.hsr.ch/APF/files/Screen_Shot_2017-11-28_at_07.30.14.png  https://wiki.ifs.hsr.ch/APF/files/Screen_Shot_2017-11-28_at_07.30.23.png  **Known Uses**  Java synchronized Block  Implementierung des *Scoped Locking* Idioms  **Konsequenzen**  Vorteile   * Erhöhte Robustheit : Durch das automatische *handling* des Locks werden Programmier-Fehler, welche zu mangelnder Synchronisation führen, vermieden.   Nachteile   * Deadlockgefahr : Wenn eine rekursive Methode das Lock verwendet, macht sie mit sich selber ein Deadlock → Abhilfe siehe **Thread Safe Interface** * Limitationen durch Sprach-spezifische Semantik : Das Pattern basiert auf C++ Features. Gewisse systemabhängige *system calls* werden nicht unterstützt. Zum Beispiel werden Locks durch *process abort* oder ''exit (thread\_exit ()) nicht freigegeben. |
| **Strategized Locking** | Implementation des Strategy-Patterns, wobei die Strategien verschiedene Synchronisierungsmechanismen darstellen. Die Applikation verwendet diese dann für das Absichern von kritischen Abschnitten, ohne genau zu wissen, welche Strategie es ist. Somit kann sie einfach ausgetauscht werden.  **Kontext**  Eine Applikation, deren Komponenten verschiedene Parallelitätsarchitekturen effizient unterstützen sollen.  **Problem**  Komponenten, auf die gleichzeitig parallel zugegriffen werden, müssen ihre "Critical sections" absichern, sodass nur ein Thread sie zur gleichen Zeit bearbeitet.   * Eine Komponente kann von verschiedenen Applikationen verwendet werden, die unterschiedliche Synchronisierungsstrategien einsetzen. Es soll möglich sein, die Synchronisierungsstrategie der Komponenten an das jeweilige Szenario anzupassen. * Verbesserungen und Bugfixes sollen in einer ganzen Komponente gleichzeitig und einfach umsetzbar sein.   **Lösung**  Kapselung der Synchronisierungsstrategie in einen Typen, der einer "lockable"-Klasse als Parameter mitgegeben werden kann. Die Klasse lockt/unlockt die Sections dann über diesen Typen, ohne genau zu wissen, was dahinter geschieht.  **Struktur**  Lock  Ein Interface oder eine abstrakte Klasse, die Methoden wie acquire&release anbietet.  Concrete locking strategie  Erbt oder implementiert "Lock". Diese Klasse implementiert eine wirkliche Synchronisierungsstrategie. Mögliche Strategien sind: recursive Mutex, non-recursive Mutex, read/write Lock, Semaphore, null-lock (Strategy-Pattern)  Guard  Für ScopedLocking wird eine Guard-Klasse definiert, die im Konstruktor ein per Parameter übergebenes Lock lockt und im Destruktor wieder freigibt. Vor jeder synchronisierten Methode wird ein Guard-Objekt instanziert und so der kritische Abschnitt synchronisiert.  **Ablauf**  https://wiki.ifs.hsr.ch/APF/files/Usage_2.jpg  **Known Uses**   * ACE: Adaptive Communication Environment Framework * ATL Wizards: Microsoft ATL Wizard in VisualStudio   **Konsequenzen**  Vorteile   * Verbesserte Flexibilität: Neue Synchronisierungsstrategien können einfach hinzugefügt werden, ohne anderen Code zu beeinflussen. * Verringerter Maintenanceaufwand: Die Implementation einer Methode gibt es nur einmal, nicht einmal pro Strategie * Wiederverwendbarkeit: Durch die Möglichkeit bei einer Komponente die Strategie auszutauschen, wird sie einfacher wiederverwendbar.   Nachteile   * Overengineering: Weil es so einfach ist, die Strategie auszutauschen, kann es vorkommen, dass ein Entwickler eine, für eine Komponente unpassende, Strategie einsetzt und es trotzdem "funktioniert |
| **Thread Safe Interface** | Das *Thread-Safe Interface* Design Pattern minimiert den Locking Overhead und stellt sicher, dass sich Methodenaufrufe innerhalb einer Komponenten nicht selber blockieren, indem sie ein Lock beziehen, das sie bereits besitzen.  **Kontext**  Komponenten in Multi-treaded Applikationen bei denen sich Methoden gegenseitig aufrufen.  **Problem**   * Klassen besitzen normalerweise mehrere public und private Methoden die sich gegenseitig aufrufen. * Non-Recursive Locks führen zu einem Self Deadlock * Um die Self Deadlocks zu verhindern, werden Recursive Mutexes benötigt. Diese haben jedoch eine unnötigen Overhead, da sie den Mutex zwischen jedem Methodenaufruf innerhalb einer Klasse immer wieder freigeben und akquirieren müssen.   **Lösung**  Alle Komponenten bei denen sich Methoden gegenseitig aufrufen, müssen nach den folgenden Kriterien strukturiert werden.   1. Interface methods check    1. Nur Interface Methods (z.B public Methods) sollen das Lock beziehen. Diese Methode leitet das Lock dann an die Implementation Methods (z.B private Methods) weiter    2. Sobald alle Methodenaufrufe abgearbeitet sind, gibt die Interface Methode das Lock wieder frei 2. Implementation methods trust    1. Die Implementation Methods müssen darauf vertrauen, dass sie nur von Methoden aufgerufen werden, die das Lock bereits bezogen haben.    2. Die Implementation Methods beziehen nie das Lock. Sie geben das Lock auch niemals frei.    3. Die Implementation Methods rufen nie die Interface Methoden auf.   **Beispiel**  https://wiki.ifs.hsr.ch/APF/files/thread-safe-interface.png  **Variants**   * *Thread-Safe Facade*: Wird verwendet wenn ein grösseres Subsystem synchronisiert werden muss. * *Thread-Safe Wrapper Facade.* Wird verwendet um unsynchronisierte Klassen zu synchronisieren. Die Facade kapselt die unsynchronisierte Klasse. Die Synchronisation findet in der Facade statt. Die Methodenaufrufe werden an die unsynchronisierte Klasse weitergeleitet.   **Known Uses**  Java Hashtable  Public Methoden wie z.B put() beziehen das Lock. Wenn die interne Datenstruktur der Hashtable gefüllt ist, muss diese erweitert werden und die Hashes neu berechnet werden. Die dafür notwendige private rehash() Methode muss dank dem *Thread-Safe Interface* Pattern nicht synchronisiert sein. Dies steigert die Performance. (Self-Deadlocks sind in Java wegen den Reentrant Monitors nicht möglich)  Java Collections  Hier kommt die *Thread-Safe Wrapper Facade* zur Anwendung. Die Collections Klasse nimmt ein beliebige Map Klasse entgegen und retourniert eine SynchronizedMap. Diese übernimmt die Synchronisation und leitet den Methodenaufruf an die herkömmliche Map weiter.  **Konsequenzen**  Vorteile   * Increased robustness: Keine Self-Deadlocks * Enhanced performance: Locks werden nich unnötig bezogen und freigegeben * Simplification of software: Locking und funktionale Implementation ist strikt getrennt   Nachteile   * Additional indirection and extra methods: Jede Interface Methode benötigt mindestens eine Implementations Methode. * Potential deadlock: Deadlocks können Auftreten wenn die Implementations Methode der Komponente A, eine Interface Methode der Komponente B aufruft, wessen Implementationsmethode dann wiederum eine Interface Methode auf A aufruft. * Potential for misuse: Private Methoden könnten direkt aufgerufen werden. * Potential overhead: Mehrere Komponenten beziehen immer ein anderes Lock. Dies verursacht einen Overhead, da das Lock nicht geteilt werden kann.   **Verwandte Patterns**   * Decorator pattern: Das *Thread-Safe Interface* Pattern fügt Locking Strategien transparent hinzu. * *Strategized Locking pattern* sollte zusammen mit dem *Thread-Safe Interface* Pattern verwendet werden. |
| **Half Sync Half Async** | Das Half-Sync/Half-Async Pattern vereinfacht das Zusammenspiel von synchronen und asynchronen Programmteilen in parallelen Umgebungen.  **Kontext**  Ein paralleles System, welches sowohl synchrone als auch asynchrone Aufgaben ausführt, die miteinander interagieren.  **Problem**  Einerseits ist asynchrone Programmierung wichtig für die Geschwindigkeit einer Applikation, andererseits wäre synchrone Programmierung einfacher und weniger fehleranfällig.  Wenn gleichzeitig synchron und asynchron programmiert werden möchte, gibt es folgende Schwierigkeiten:   * Einfache Applikationen sollen weiterhin synchron geschrieben werden können; gleichzeitig sollte Asynchronität für Performante Applikationen bereitstehen. * Die synchronen und asynchronen Programmteile sollten ohne Performance-Verluste kommunizieren können.   **Lösung**  Aufteilen der Dienste in zwei Layer, *synchronous* und *asynchronous*, sowie Hinzufügen eines *queueing*-Layers dazwischen zwecks Kommunikation.   * Auslagern von Higher-Layer Problemen (wie Datenbankabfragen und Dateitransfers) synchron in eigene Threads. Diese haben oft eine lange Laufzeit. * Auslagern von Lower-Layer Problemen (wie Protokoll-Handler und Treiber) asynchron zur Performancesteigerung. * Higher- und Lower-Layer Applikationsteile kommunizieren mit Messages über einen Queing-Layer   **Struktur**  Die Struktur folgt dem Layers Pattern (POSA 1)  https://wiki.ifs.hsr.ch/APF/files/hsha-domain.png  **Ablauf**  https://wiki.ifs.hsr.ch/APF/files/hsha-sequence.png   1. Asynchrone Phase    * Eine externe Quelle (z.B. Netzwerk Interface) produziert einen Event und notifiziert den asynchronen Layer über Interrupts    * Der asynchrone Layer verarbeitet diesen und stellt eine entsprechende Message in die Queue 2. Queueing Phase    * Der Queuing Layer notifiziert den synchronen Layer über die Message    * Der Queuing Layer buffered Messages zwischen den beiden Layer    * Der Queuing Layer implementiert das Mediator Pattern ([GoF](https://wiki.ifs.hsr.ch/APF/wiki.cgi?GoF&printerfriendly)) 3. Sychrone Phase    * Der entsprechende synchrone Service holt die Message ab und verarbeitet sie   **Known Uses**  UNIX Networking Subsystems  Nutzen dieses Pattern zum Behandlen von I/O.  Restaurant  In Restaurants ist der Gastgeber für die Begrüssung der Gäste verantwortlich. Ebenfalls entscheidet er wann ein Gast einen Tisch bekommt oder ob er momentan in der Warteschlange auf einen freien Tisch warten muss. Der Gastgeber wird von allen Gästen "geteilt" und kann daher nicht viel Zeit mit einer bestimmten Partei verbringen. Nachdem die Gäste an einem Tisch Platz genommen haben, widmet sich eine Kellnerin dem Service an diesem Tisch.  **Konsequenzen**  Vorteile   * **Einfachheit** und **Perfomance**: Einfacherer Programmierung, ohne grosse Performance-Verluste. Es werden keine Sychronisationsprimitiven benötigt. * **Entkopplung** von asynchronen und synchronen Applikationsteilen * **Zentralisierung der Kommunikation** weil Messages über den Queueing-Layer versendet werden.   Nachteile   * **Performance**: Bei der Kommunikation zwischen synchronen und asynchronen Applikationsteilen gibt es einen Overhead, auch bedingt durch das Context Switching * Unter Umständen **wenige Vorteile** für Higher-Level Applikationen, da diese von der Trennung nicht profitieren können. * **Schwieriges Debugging**: Da Abläufe nicht mehr so stark gekoppelt sind, wird das Debuggen schwieriger.   **Verwandte Patterns**   * **Layers Pattern** (POSA1) wird zur Trennung der verschiedenen Schichten verwendet.   Ähnliche Patterns:   * **Proactor Pattern**: Geeigneter für Entwickler, welche mit Asynchronität vertraut sind. * **Reactor Pattern** mit dem **Active Object Pattern** erfüllt einen ähnlichen Zweck * **Leader/Follower Pattern**, dieser hat aber keinen Kommunikationslayer. * **Pipes and Filters Pattern (POSA1)**, Producer/Consumer Verhalten zwischen synchronous und asynchronous Layer |
| **Leader Followers** | Das 'Leaders/Followers' Pattern ist ein effizientes Concurrency-Modell bei dem mehrere Threads sich abwechslungsweise um Eventverarbeitung kümmern.  **Kontext**  Eine event-getriebene Applikation, bei der mehrere Service Requests von verschiedenen Eventquellen parallel eintreffen können. Diese Anfragen müssen effizient verarbeitet werden können. Die zuständigen Threads teilen sich die Eventquellen.  **Problem**  Problem: Multithreading wird oft eingesetzt um gleichzeitig Events verarbeiten zu können. Es ist allerdings nicht trivial, eine highperformance Multithreaded Applikation zu entwickeln.   * Service Requests können von mehreren Eventquellen eintreffen. Es wird also ein gut performendes Demultiplexing benötigt. Eine fixe Zuteilung der Threads pro Event Source ist keine Option. * Für maximale Performance muss Nebenläufigkeitsoverhead so klein wie möglich gehalten werden (context-switching, synchronisierung) * Mehrere Threads demultiplexieren Events von einer geteilten Menge an Eventquellen. Um Raceconditions zu verhindern, müssen sich die Threads untereinander koordinieren.   **Lösung**  Ein Threadpool überwacht mehrere Eventquellen, wobei immer nur ein Thread (der Leader) einen eintreffenden Event empfängt und danach diesen synchron an einen entsprechenden Eventhandler dispatcht. Vorher ernennt er einen anderen Thread (Followers) zum Leader, um weitere Events zu empfangen.  **Struktur**  Handle   * Betriebssystem Handle die eine Eventquelle identifizieren (z.B Netzwerkverbindung, Open File) * Kann mehrere Events einreihen   Handle SetEnthält mehrere Eventquellen  Event HandlerInterface mit mehreren Hook Methoden für applikationsspezifisches Event Processing  Thread Pool Gruppe von Threads die einen Synchronizer teilen (z.b Semaphore oder Condition Variable)  **Ablauf**   1. Leader thread demultiplexing    1. Leader Thread wartet auf einen Event der in einem beliebigen Handle im Handle Set auftritt.    2. Wenn kein Leader vorhanden ist, werden die Events vom OS in eine Queue gestellt 2. Follower thread promotion    1. Der Thread Pool Synchronizer wählt einen Follower aus dem Thread Pool zum neuen Leader 3. Event handler demultiplexing and event processing    1. Der ehemalige Leader spielt nun die Rolle des Processing Threads    2. Der Processing Thread demultiplexiert den Event nebenläufig an den Event Handler    3. Der Processing Thread kann nebenläufig zum aktuellen Leader und anderen Processing Threads laufen. 4. Rejoining the thread pool    1. Wurde ein Event bearbeitet reiht sich der Processing Thread wieder als Follower ein.    2. Wenn kein anderen Leader vorhanden ist, wird der Processing Thread direkt wieder ein Leader.   https://wiki.ifs.hsr.ch/APF/files/leader-followers-sequence.jpg  **Known Uses**   * ACE Framework * CORBA   **Real Life Example**  In der Nacht sind alle Taxifahrer am schlafen bis auf einen der auf Anrufe wartet. Wenn ein Gast das Taxi beansprucht, wird dieser Taxifahrer den Gast bediehenen. Bevor er los fährt, weckt er den nächsten Fahrer auf, der dann wiederum auf den nächsten Anruf wartet.  **Konsequenzen**  Vorteile   * **Performanceverbesserung** gegenüber Half-Sync/Half-Reactive-Ansatz (CPU, Memory allocation und weniger locking, priority inversion und kontext switches). * **Einfache Programmierung** des concurrency Modells für asynchrone requests/responses und demultiplexierung.   Nachteile   * **Komplexe implementation**: durch die Parallelität kann die Implementation komplex werden. * **Unflexibel**: Es ist schwierig, dieses Pattern z.B. mit einer Priorisierung zu erweitern, da es keine explizite Queue gibt. * **Netzwerk I/O bottleneck**: Nur ein einziger thread kann jeweils demultiplexieren.   **Verwandte Patterns**   * [U11\_2\_Reactor\_HS17](https://wiki.ifs.hsr.ch/APF/wiki.cgi?U11_2_Reactor_HS17&printerfriendly) als Alternative, wenn nur kurze Verarbeitungszeiten nötig sind. * [U11\_2\_Proactor\_HS17](https://wiki.ifs.hsr.ch/APF/wiki.cgi?U11_2_Proactor_HS17&printerfriendly) kann sehr ähnlich wie das Leader Followers Pattern genutzt werden. * [U11\_2\_HalfSyncHalfAsync\_HS17](https://wiki.ifs.hsr.ch/APF/wiki.cgi?U11_2_HalfSyncHalfAsync_HS17&printerfriendly) und [U11\_2\_ActiveObject\_HS17](https://wiki.ifs.hsr.ch/APF/wiki.cgi?U11_2_ActiveObject_HS17&printerfriendly) sind zusammen eine alternative Implementierung * Das *Controlled Reactor pattern* ist eine Performance-verbesserte Version des Rector patterns. |

|  |  |
| --- | --- |
| **Fault Tolerant Software Patterns** | |
| **Introduction** | The goal of the book is to show patterns, which help designing a fault tolerant system, or in other words, a system that operates correctly, although some parts aren't. This chapter gives an overview of the terms used.  1. Difference Fault/Failure/Error   * Failure is a system behaviour that does not comply with the system specification. Failures are caused by errors. * Error is a system behaviour from where a failure can happen. They have two types (timing or value). Errors can be detected before a system failure happens. They are manifestations of faults. * Fault is the defect in a system, which can cause an error. It is often called a bug. Neither the software nor the observer notice a fault until an error occurs. A fault can be latent (in a dormant state) or active (an error occurs). Multiple faults can cause the same error or be part of the error.   2. Type of failures   * A fail-silent occurs, when the failing part presents the correct result or none at all. * A crash failure occurs, when the failing part stops after a fail-silent. * A fail-stop is a visible crash failure (i.e. the system realizes the part failed). * Failure consistency A failure is consistent, if it appears the same for all observers. An inconsistent failure appears differently to each observer. They are also called two-faced or malicious.   3. Single faults  A wide-spread assumption is that only one error occurs at a time and the recovery from that is finished before another error occurs. It is further assumed that errors are independent of each other. Table 1.1 shows the number of redundant units for a failure tolerant system.  https://wiki.ifs.hsr.ch/APF/files/Selection_001.png  4. Coverage  The coverage is the conditional probability that the system will recover automatically within the required time interval given that an error has occurred.  coverage = Cond\_prob(successful automatic recovery within time | error has occurred)  Coverage can be calculated from the probability associated with detection and recovery.  coverage = Prob (successful error detection) \* Prob (successful error recovery)    5. Reliability  The reliability of a system is the probability that it will perform without deviations from the agreed-upon behavior for a specific period of time. This means no failures occur during a specified time.  reliability = exp(-(1/MTTF))  MTTF = Mean Time To Failure  6. Availability  A system’s availability is the percentage of time that it can perform its designed function. Uptime is when the system is available, downtime is when it is not.  availability = MTTF/(MTTF+MTTR)  MTTR = Mean Time To Repair    Sekunden/Stunde 3600 s/h  Sekunden/Tag 86'400 s/d  Stunden/Jahr 8760 h/y  Minuten/Jahr 525'600 m/y  7. Dependability  Dependability is the measure of a system's trustworthiness. (reliability/availability/safety/security)  8. Performance and Reliablility  These are two closely related concepts. When a system does not meet the performance requirement, reliability can also be impacted. For example, the performance criteria is to handle 1 Million requests. What happens, when more than that want to connect? Is the service no longer available? Or does it operate with reduced performance?  A well-designed fault tolerant system will be able to process the required requests and gracefully handle excess workload. |
| **Fault Tolerant Mindset** | Applying a Fault Tolerant Mindset to all stages of software development is essential and basically means always asking 'What can go wrong?'. The different techniques to design for fault tolerance will be covered in this chapter.  1. Design Tradeoffs  'Every problem in computer science boils down to tradeoffs' – Professor L. J. Henschen  MTTF and MTTR determine the reliability and availability of a system and are an example of a common tradeoff. Sometimes, MTTF is more important (no failures during Space Shuttle mission) and other times MTTR must be very short (fixing errors in telecommunications systems).  2. Quality vs. Fault Tolerance  'Good quality processes guarantee fault tolerance'. This is a common misconception, because even though the two ideas are related, they are distinct from each other.  *Fault tolerance*: The ability of the system to execute properly even though there are faults present.  *Quality*: Refers to how fault-free a system is.  That means a high quality system will have fewer faults than a lower quality system. But that doesn't affect the system's ability to deal with the remaining faults.  3. Keep It Simple  'KIS' is a well known principle of fault tolerant design. The more complex a system becomes the harder it is to ensure correct operation, because additional complexity requires writing additional code, which in turn means that there will be more faults present. That's why one common goal of KIS is to reduce the number of code lines dedicated to a task and to build just the right system, without any extra features.  4. Defensive Programming Techniques  Defensive Programming builds on the Fault Tolerant Mindset to produce code that defends itself against potential problems and is done by constantly asking 'What can go wrong here?', 'How might this fail?'. Here are a few things to keep in mind:  5. Faults in Fault Tolerance Code  Even the error handling code can have faults and adding complexity to it also increases the risk of adding faults. That's why we 'Keep It Simple'.  6. Memory Corruption  Never assume that data storage is always correct. Stray writes, bad memory chips, transient errors, incorrect usage of pointers, etc. can invalidate information. That's why it should be checked before it's used, especially when it is used to determine program execution paths.  7. Design for Maintainability  Fault tolerant systems usually last a long time. Design them in a way that will make the life of the maintainer easier, because they might not be one of the initial designers.  8. Coding Standards  Coding standards are a simple but effective way of improving software quality. But always keep in mind: People cannot remember a large number of things. Having a small number of standards is much more effective than having 100 of them.  9. The Role of Verification  Testing and verification are essential when creating fault tolerant systems, because they ensure that fault prevention and quality efforts are successful. They also provide the data needed to compute the expected reliability of a system. A very useful method is operational profile testing. An operational profile describes the usage of the system in quantitative terms and the most typical scenarios that it will process. This information helps define the most appropriate tests to be run and how to focus testing efforts.  10. Fault Insertion Testing  Fault insertion testing is a technique where testers insert known faults into a system and monitor its behaviour. This type of testing serves two purposes:   1. Identifying faults in the system's error handling processes. 2. Providing data for the computation of coverage factors (successful vs. unsuccessful recovery).   11. Fault Tolerant Design Methodology  The following six-step methodology is useful to build upon the Fault Tolerant Mindset and produce highly available systems:   1. Assess what can go wrong with the system. 2. After identifying the risks, define strategies to mitigate them. 3. Identify the primary system dividing points and modes of redundancy. 4. Now the architectural and major design decisions that influence the whole system can be made. 5. Design in the capabilities for the system to implement the risk mitigation strategies (step 2). 6. No matter how fault tolerant a system is, it almost always must be managed and administered by someone. Designing the required human computer interactions and modes of management is important and must ensure that failures won't be caused by the ones trying to administer the system. |
| **Introduction to the patterns** | The execution time lifecycle of a fault is described by the four phases shown in the picture below:  *Error detection*  The first thing that must happen when a fault activates and an error occurs, is error detection. Once detected, the error must be processed, which is done in one of the next two phases. Since they are executed in real time, they will affect the availability of the system.  *Error recovery*  In this phase, the erroneous system state is replaced by an error-free system state, from which the system can attempt to resume normal operation.  *Error mitigation*  In some cases, instead of transitioning to a different system state, the error can be removed or mitigated (= gelindert, abgeschwächt). For example, by correcting an erroneous data value.  *Fault treatment*  Fault treatment is the step where the fault that caused the error is removed from the system through update or patching mechanisms.  https://wiki.ifs.hsr.ch/APF/files/4phases.png  The patterns that follow in the later chapters discuss fault tolerance both to protect data from the effects of faults and to protect execution flows from faults. The choice of which technique is appropriate for any given design problem is based on the context of the problem and the system's specifications.  1. Shared Context for These Patterns  The shared context for these patterns include the following attributes:  1.1 Real-Time  The patterns apply to both soft and hard real-time systems, but hard real-time systems have much more stringent constraints than soft real-time systems.  *Soft real-time*  No functions that absolutely must occur at certain times or with a specified frequency. E.g. responses from a web server are desired very quickly, but if some are a bit slower it won't result in a system failure.  *Hard real-time*  Catastrophic consequences if a deadline is missed. E.g. late results in an aircraft flight control system must definitely be prevented.  1.2 High Reliability  The system will have stringent availability and correctness requirements.  *High Availability*  The system should be available for service as much as possible. Its requirements will state the customer's expectations.  *Failure Rate Requirements*  The system has requirements for acceptable failure rates (e.g. only 1 in 100'000 transactions may be mishandled).  1.3 State or Stateless  The system state is the combination of local variables, execution location (i.e. the program counter) and workload status that describes what the system is doing at any specific instance. Systems that have only small amounts of state information (e.g. a web server) have different needs for recovery techniques than systems that keep large amounts of state information (e.g. a computer's operating system). There are patterns that address the needs of both stateless and stateful systems and other patterns that are more appropriate to one of them.  1.4 External Observers  The ability to provide information about all detected errors, failures and any actions taken to compensate for them is a basic requirement on a fault tolerant system.  1.5 Fault Tolerance is Not Free  Achieving fault tolerance and maintaining a state of readiness to handle errors and failures is not free. Both at design and execution time.  1.6 Long Lived Systems  Fault tolerant systems have a longer life than other systems, because they don't fail in ways that prompt their replacement and because they are costlier to build (owners want to keep them for a long time). |

|  |  |
| --- | --- |
| **Architectural pattern** | |
| **Language Map** |  |
| **Units of Mitigation** | 1. Problem  There will be faults. These faults will diminish service or take out your system completely. How do you reduce or mitigate risk of complete shutdown?  2. Solution  Divide system into *Units of Mitigation*. One unit is the smallest part that can reliably detect errors and recover from failure state on its own. Unit size always depends on specific system. Sometimes it's even the whole system itself.  Some patterns can only be applied if the system is divided into smaller units  There need to be clear and strict boundaries between units, so that errors do not propagate. This makes error handling easier.  Once units are defined, certain patterns like redundancy or failover can be applied to each unit as needed.  3. Advantages   * Reduce recovery time * Easier, well-defined error handling   4. Drawbacks   * Needs strict boundaries between units. |
| **Escalation** | 1. Problem   * Trouble with processing or mitigating errors or failures in a component. Error processing does not run to completion but needs to in order to resume normal operation. * CorrectingAudits, RollBack and RollForward have not worked. * The system is build around MinimizeHumanIntervention and SomeoneInCharge.   2. Solution  When recovery or mitigation is failing, escalate the action to the next more drastic action.  If a component gets stuck because of a transient - i.e. a peer sends erroneous messages - RidingOverTransients should be used.  Generally, LimitRetries is a good way to get out of a loop. Sometimes it is necessary to de-localize error handling. If first attempts do not result in success, recover state from a CheckPoint and RollBack. If that does not help, Restart the component. Failover to a redundant processor in case restarting does not solve the problem.  When escalating, SomeoneInCharge needs to be notified for it to trigger the next action. When human intervention becomes necessary, there should be a list of actions that do not have a big impact on the system and are likely to solve the problem. If that does not work you know the drill.  3. The end of the line  If all else fails, sometimes it is best to isolate the faulty part and let the healthy part run independently as best as it can. This partial degradation is an option if the faulty components can easily be confined. For example, using an aproximated value if the exact value can't be calculated.  4. Advantages   * If a recovery step fails or gets stuck, there is always a fallback.   5. Drawbacks   * It requires a lot of thinking and experience to design a proper escalation plan |
| **Correcting Audits** | 1. Data   * Static Doesn't change very often (example: Configuration data) * Dynamic Changes frequently (example: per transaction, per request, currency exchange rates)   Data errors may occur because something corrupted the data (example: low level hardware, alpha rays)  2. Problem:  Errors from faulty data easily propagate throughout the system and there is a chance, that another task uses the falsy data and makes a possibly failing decision with it. The context has also a meaning (1974 is valid as a year but not as an age). This helps to define allowable data checks.  2.1 Types of caused errors:   * Future computations might be incorrect (example: incorrect exchange rate leads to incorrect conversions) * Execution transfers incorrectly (example: wrong data in switch statement leads to other code execution) * Faulty data causes errors in other unrelated data (pointer that overwrites an other part) * Orphaned resources (pointer on data and pointer gets deleted)   2.2 Data checks   * structural properties Are linked lists correctly linked, stack and queue pointers within bounds, number of items in a container the actual number of items. * Known correlations The same or related data is stored elsewhere, known conversion factors between values, cross linkages between different structures * Sanity checks Values are in an expected range, checksums are correct * Direct comparison Duplicate copies of data are stored to serve as checks to each other. (More for static data than dynamic)   3. Solution:  Design the system's data to be easily audited. Use two way linked lists, because linkages are redundant and enable easy checking. Use redundant locations in other parts of the system. Use non-trivial values for static configuration values. This leads to a system where many data errors can be corrected automatically. And therefore the time from detection to error-free operation is reduced -> better availability  3.1 Correction methods  Normally the data can be corrected with the redundant information. When this is the case the system can resume at the point of detection. This emphasizes the need to put the detection mechanism as close to the point fo first access as possible. Maybe a rollback is required or if it is impossible to return to that point, return to a reference point.  Where there's one data error there might be others and it is to consider what other data might be erroneous. All the data that is important to the systems operation should be checked. For each data structure in the system you should consider what would happen if it became corrupted and erroneous. In order to detect the fault that is causing the erroneous data to appear, the data error needs to be recorded and tracked. Any detection of data errors should be logged to enable for causal analysis.  4. Points to take with you  Three tasks that should occur when data errors are detected:   1. correction, 2. logging, 3. resume execution with the corrected data.   This can be done by three different tasks or one part of the system designed to perform these tasks as quickly as possible. |
| **Redundancy** | 1. Problem  The main goal is to design a system that has the maximum possible uptime. In other words, how can we reduce time between error detection and resumption of normal operation?  2. Solution  *spatial redundancy (räumliche Redundanz) :*   * In terms of hardware, there are multiple copies of hardware. The hardware doesn't have to be identical. It is only required to perform same computations. E.g. In Space Shuttle 4 computers were used to perform same critical function. Afterwards the results were compared. * Since software is deterministic, multiple instances of identical software will all fail when the same stimulus is given. * E.g. In avionics systems a way of providing software redundancy is to provide for transition periods where some processors are executing on the new versions of software, and other processors are still executing on the old version. After a time of testing the new versions and the processors with old version be upgraded to the new version   *temporal redundancy:*   * Temporal redundancy refers to redundancy that occurs over time, which helps achieve correct results but lengthens the time of unavailability. * RecoveryBlocks provides a temporal software redundancy   *informational redundancy:*   * It is provided by having multiple versions of the same data available to the system. The versions can be stored in different places on different types of storage. CorrectingAudits can use to quickly make a correction.   https://wiki.ifs.hsr.ch/APF/files/Screenshot_from_2017-10-12_12-01-20.png  *Active-Active*  Both of the elements are active. Either of the elements is capable of processing the entire work load Active-active provides the fastest recovery, but it has a high cost.  *Active-Standby*  The standby element is not performing useful work. In other words the standby has a different state than active element. The more often they share state the quicker the recovery.  *N+M*  There are N active elements processing the workload. There are also M redundant, standby elements in the system. This requires that there be some way to redirect traffic from any of the N active units to one of the redundant sets.  3. Advantages   * immediate instantiation of the error-free state * reduction of unavailability   4. Drawbacks   * costs of extra resources such as memory or processing * risks and costs associated with maintaining the redundant elements |
| **Recovery blocks** | 1. Problem  Software will always respond in the same way if it is in the same state and encounters the same input. So how can we recover from an error, if repeatedly executing the same code with the same input will always result in an error?  2. Solution  By using N-Version Programming (NVP).  At system level, NVP means that multiple independent development teams provide multiple implementations that satisfy the same specification. At the scale of one development team, they can produce multiple versions of individual modules or algorithms. The goal is to make the different versions resistant to the same fault stimuli.  The system can then execute these redundant implementations simultaneously and pick the best answer using a combination of redundancy and voting. Another way of using them is to execute them sequentially by calling the next alternative when the system detects errors. Each of the alternatives is a so called recovery block. A program containing recovery blocks consists of one primary block and multiple secondary blocks. If the result of the primary block fails its acceptance test, then the secondary blocks are executed in sequence until the acceptance test passes.  https://wiki.ifs.hsr.ch/APF/files/sequential.png  A common way of using recovery blocks is to make each successive block more simple than its predecessor, so that each has a higher probability of satisfying the acceptance test. The downside of this method is that some information is lost at each successive block because it is not performing all of the initial actions.  Another way would be to use entirely different algorithms that produce the same result (e.g. sorting algorithms). The downside of this method is the usual lack of alternative algorithms.  https://wiki.ifs.hsr.ch/APF/files/sorting.png  3. Things to keep in mind   * The time spent processing the same information in successive secondary blocks is time that the system is unavailable for other work. * The acceptance tests and secondary blocks add complexity to the system.   So, avoid producing too many secondary blocks and try to balance the need to have the result with the overall system availability specification. |
| **Someone in charge** | 1. Prerequisites  You are designing a fault tolerant system. The system is built on Units of Mitigation and Redundancy System. Furthermore, it is designed to process errors automatically using Minimize Human Intervention  2. Problem  Error processing is not trivial. Anything can go wrong even during error processing. If not handled correctly, the system may stop handling errors on top of not resuming normal functionality. Both detection and recovery are hard to do.  3. Solution  If there is at least one part that knows what the system is supposed to do and observes its behaviour, it can act on that knowledge and bring the system back on track, thus making it more robust. That component is then *Someone In Charge*. *Someone In Charge* might be responsible for recovery itself, or it can notify a responsible part, which will handle things further.  Both the Fault Observer and the System Monitor can perform this task, although they have different primary responsibilities. The Fault Observer collects and distributes error reports, the System Monitor detects errors.  When using redundancy, there will be at least one active element. It will be in charge of error processing on the other elements. If there are multiple active ones, decide on ONE. I.e. by lowest ID etc.  Of course, there is not one global instance of *Someone In Charge*. For every part in the system (Unit Of Mitigation) there should be *Someone In Charge* of error processing as close as possible to where they are needed. This makes sense for multiple reasons:   * There is not one point of failure * A global error processor will become complex very quickly, increasing the potential for faults   If a given action is not fruitful, Someone In Charge can Escalate to more drastic measures.  Monitoring error processing ensures that the system will not silently fail. Techniques to determine this are for example Heartbeats and Acknowledgements (as in TCP/IP).  4. Summary  A fault tolerant system should have Someone In Charge to monitor the health of a system and to take or delegate action if an error occurs. It should furthermore monitor error processing, making sure the system does not fail silently. |
| **Minimize Human Intervention** | 1. Problem  There are three categories of errors in highly available systems. These are hardware, software and procedural. Procedural errors are the result of mistakes made by operating personnel.  Usually people do worse than computers in situations where multiple steps is required to accomplish a task. People have a tendency to skip steps. Human operators get bored and they don't pay attention anymore to routine, monotonous tasks.  2. Solution   * One of the things that people do better than machines is interpreting situation from events that seem unrelated. In this case Experienced people can solve problems swiftly. Therefore, the system should provide experienced people enough information. (See for further info MaximizeHumanParticipation) * There is an old saying : ‘A quiet system is a dead system.’ For example the spinning wheel icons in operating systems gives the user an indication that the system is still working. That's why the system must provide the operating personnel enough information so that they can be calm. Otherwise they will do something that could worsen the situation. * Design the system that it is able to process and resolve errors automatically. This speeds up error recovery and reduces the risk of procedural errors. The system should provide the operating personnel enough information, so they can follow the progress of error detection & processing. But the operating personnel should not be needed for the resolution of errors. Make sure that errors are reported to the FaultObserver. |
| **Maximaze Human Participation** | 1. Problem  As described in MinimizeHumanIntervention, the involvement of people should be limited.  But faults exist in the system. People built the system. The designers of the system understand how it works, and how it should work. Experienced operators know how it operates. The expertise of these people can help to resolve problems quickly.  2. Solution   * People tend to have a broader view of the situation. They can make up for gaps in the programming that keeps the system from a clear view, and they can know about physical problems, for example that the system backups are a long distance away. * The system provides commands for experts to resolve errors. With commands the expert is able to steer the system's recovery. Because only the appropriate commands are displayed, it eliminates false attempts and narrows down to productive paths. * There is no one always-correct kind of involvement. For some systems, such as safety-critical systems like avionics, the ability of the operator to override or alter error processing is essential. In other systems, such as home appliances, the operator standing in front of the microwave oven will be unable to guide the oven’s error processing at all. * The system can enter a ‘safe mode’ and perform no further automatic actions while waiting for the human participation. This is the correct thing to do in situations where the risk of incorrect action is very high, for example spacecraft. * Know the user and their ability. Design the system to enable knowledgeable operating personnel to participate in a positive way. Provide appropriate MaintenanceInterface and FaultObserver capabilities to give the operators the information that they need to be able to contribute constructively. |
| **Maintenance Interface** | 1. Problem  Many highly available systems have maintenance commands to perform configuration, logging or health and status checking actions. If the system provides only one interface, then the normal and maintenance inputs will be intermixed and will need to be separated again inside the system, using its processing resources.  This is a problem because when it's most important to enter maintenance commands might be when the system is overloaded with application inputs and losing the few maintenance commands within all the others is quite likely.  Intermixing both input types might also cause security problems, because people with access to the application will be able to submit maintenance commands.  2. Solution  A simple but effective solution to both of the above problems is to have a separate maintenance interface. Some advantages of this method are:   * Enables the usage of priority levels (higher priority for maintenance commands) * People with access to the application lose the ability to submit maintenance commands * Bundles all the critical information at one place (saves time)   The maintenance interface can be used for maintenance and maintenance-like functions. Accessing logging information would be an example of a maintenance-like function, because it is not directly maintenance related, but would still be an acceptable use of the maintenance interface. That means, when defining the list of permitted uses of the maintenance channel, one should take a broad view and enable many uses. But don't forget to weigh this against the ability to get important information out quickly and also don't enable application-related workload to use it. |
| **Fault Observer** | 1. Problem:  The fault tolerant system does not stop when errors are detected and automatically corrects them. How will people know what faults and errors have been detected and processed?  2. Solution:  With the publisher-subscriber pattern, we can design components which send their errors and faults to the registered users and systems. But when we do this in every component which can detect and correct an error, we would have a lot of duplicated code, which is a very good place for other faults to hide. It can also be that the error gets reported from multiple observers and therefore reported multiple times. But this is agains the maximize the human interaction pattern (from earlier).  3. Observer:  Publishes Information to his subscribers and this happens over the maintenance interface, so that the reports solely happen over this interface and don't get mixed with "normal" information. There can be one or multiple ones, but then you have to be sure which observer reports with faults, so that the double reporting of errors doesn't happen. The interested parties are usually interested in faults and errors, since a single error can be produced by multiple faults. This helps to correlate errors and faults and to improve the fault tolerance. The observer doesn't have to be solely internal, it can also be external and receive the reports, which will be sent to the subscriber.  https://wiki.ifs.hsr.ch/APF/files/faultobserver13.PNG  4. Rembember:  Report all errors to the fault observer. It will ensure that all interested parties receive information about the errors that are occurring.  Since it gets information about all errors, it makes sense to have it perform some other fault tolerant functions as well such as someone in charge (earlier pattern).  4.1 A note about log files:  Ensure that the system has a strategy to handle the situation when the log files pile up. Two possible alternatives are to use a circular log or a set of logs that are cycled between. |
| **Detection pattern** | |
| **Language Map** | https://wiki.ifs.hsr.ch/APF/files/detection_patterns_u11.PNG |
| **Introduction** | Until now we covered architectural patterns. This next weeks we will look into detection patterns.  The first phase of fault tolerance is detection. There are two main concepts which drive the detection at execution time.   * **errors, respectively a priori knowledge**: Uses constraints that are known in advance to determine if some deviation from the normal situation of correctness exists. This includes system states, results, and any side effects.   versus   * **failures, respectively comparison of redundant elements**: Comparison of redundant systems. There must be enough information present, either in the values to be compared or the context to enable the identification of the faulty component. Just determine that something is faulty, doesn't help.   Another aspect of detection is the adding of new techniques that help a system to learn about correct system behaviour. The system can keep a record of errors and faults that are detected and apply Bayesian learning techniques to learn to automatically detect errors and failures.  The system must be able to detect failures and errors. It's more desirable to detect an error than it is to detect a failure because errors are not visible to the observers of the system. Two mechanisms for detecting errors are in common use.   * test function return codes to determine if the function has returned reporting an error condition. * use the exceptions or try/catch capabilities.   They supplement the capabilities discussed by the patterns in this chapter. |
| **Fault correlation** | 1. Problem and Overview  An error or failure has been detected. There are a number of potential causes for it. In order to process an error or failure the system needs to identify the error and what fault caused the error.  Errors can be caused by any of several faults. Particular errors keep occurring. The system may ignore the error or initiates error processing.  What has the error done? Has execution stopped? If so, what capabilities are no longer available? What was the size of the stack at the time of the error? Were logs collected? What data is incorrect? Is it frequently changing data or is it a constant? Identification of the fault allows targeted recovery actions to be taken.  Fault tolerance is about handling the unanticipated and undetectable errors that occur during execution. But as faults are being removed from the system during design and test, different common errors will have been uncovered, isolated, and corrected. The clues to error types, or their signatures, learned during these activities help you know what kinds of errors are likely to occur during normal execution.  2. Best practices:   * The identified error must be masked so that the fault is tolerated. Identifying precisely what fault created the errors is needed for treatment, but not for error processing. Similar errors can be given similar error processing treatments. * Errors can occur in clumps. One fault triggers an error that in turn triggers other errors. Failure can be prevented ideally by processing the initial fault that started the chain of errors. * When several errors occur close together in time, they might be related. They should be used to triangulate the location of the error. * When data errors are detected the correlation involves the identification of related data that also needs to be checked.   3. Take Home Message:  Look at the unique signature of the error to sort it into the fault category for which error processing steps ae known. |
| **Error containment barrier** | 1. Problem  Errors in one part of the system can spread and cause errors or failures at other places in the system. Errors spread through several mechanisms: erroneous messages, corrupted (incorrect) pooled memory or actions based on the results of other incorrect actions.  So what is the first thing that the system must do when it detects an error?  2. Solution  The system should isolate the error and stop the flow of errors from one part of the system to another by using some kind of barrier.  In software this barrier is the boundary of the Unit of Mitigation that contains the fail-silent component. Failing silently implies that any error that is detected within that Unit of Mitigation presents a silent interface to other parts of the system. That means the actual internal errors are masked to provide only an indication of silent failure. When this is detected the signs and symptoms of the detected error are sent to Fault Correlation which analyzes the error and provides guidance to the techniques most useful to process the error.  If the faulty component is a hardware component then it can be isolated by activity bits that indicate whether the component is in service, error processing or some other state. One of the most important things when isolation faulty hardware is to prevent it from spreading malicious information across the system (so-called babbling problem). This can be done by isolating the network connections or other communication links to the faulty component or by marking the erroneous data to contain its future use. |
| **Complete parameter check** | 1. Problem  Detecting errors is an important task of a fault tolerant system. But the further away from the point of error, when a fault activates, that detection occurs, the larger the window in which it could cause a system failure or propagate to other parts of the system. That means it is obviously very important to detect errors as fast as possible, to keep them from spreading and harming the system.  So how can the time from fault activation to error detection be minimized?  2. Solution  There are many methods of achieving that and most of them involve frequently checking for errors.  The fastest option is to check for errors at every operation the system conducts, for example by using an active-active redundant pair of processors. This tight coupling works best for hardware failures and not as well for software failures because both processors execute the same program and will both compute an erroneous result from the same incorrect input.  Another option is to only compare the end results and not every single operation. Detection will be a bit slower, but this allows for different software implementations (NVP) to be used to compute the results.  Even without redundant results to compare there are many things that can be checked. For example whether the value of a function/method argument is within an acceptable range (array index, date, string length, etc.). One way of doing that is by using the well known method 'programming by contract', where a 'contract' specifies the obligations (input) and benefits (output) of a function/method. These contracts are implemented by using checks before and after the processing to ensure that both the input and output meet the contracted specification.  3. In Short  The quintessence of all this is: Perform frequent checks on data and operations to detect errors quickly and prevent them from propagating to the rest of the system. |
| **Existing metrics** | 1. Problem   * Overload means when more resources are needed than there are resource available to process the workload. Depending of system design the overload can take different forms. * New indicators can be created to measure the severity of the overload. With these indicators the system can be measured accurately. But the indicators bring also additional overhead into the system. If the system is already overloaded then it is a bad idea to have additional overhead around. * You can use a special software, which will work only when a overload occurs. But this option is risky because there might be latent faults in special software. It would be a bad timing for a error when a overload occurs.   2. Solution   * The operating system and other infrastructure parts provide already a range of built-in measurements. The system can use this existing metrics as indicators. For example CPU idle time could be used. It does not increase overhead, since the data is already available. * These existing metrics might not contain the exact information that is needed to control the system’s overload. Instead of basing control decisions on the measure of average time spent to handle each request, the percentage CPU occupancy can be used. It doesn’t directly report the time per request, but it offers only an indication of the total system workload. * Heartbeat and Acknowledgement are regular messages that can be expected in a system. The amount of time that they take to travel across the system can serve as an indicator of messaging path congestion.   3. Summary  Use pre-existing indicators already tied to the resource as an indicator of the system’s overload condition. |
| **Voting** | 1. Prerequisites   * Redundancy that provides multiple answers exists in the system (more than 1 running element) * Reason for redundancy is latent faults that prevent system from running error-free   2. Problem  The system has multiple answers. How does it determine the right one?  3. Solution  There needs to be some algorithm in place to find an answer. This can be done by voting. Each redundant element returns its answer, and Someone In Charge determines what is most likely the correct answer.  There exists multiple ways to conduct voting. Some of them are listed below.   * Exact Voting   + There is only one correct answer. (i.e. integer, character)   + Majority Voting     - The correct answer is decided by simple vote of majority. It is important that the number of elements is uneven, lest there be an undecided vote (i.e. 4 participants; 2 vote yes; 2 vote no)   + Generalized Median Voting     - The right result is chosen by finding the median of all votes.   + Formalized Plurality Voting     - All votes are classified (usually into intervals) and of the class with the most votes one vote is picked at random. * Inexact Voting   + There may be multiple correct answers. There is no direct comparison possible (i.e. float, double)   + Weighted Average Voting     - It is used to lessen the impact of erroneous values and is a type of non-adaptive voting where each result is weighted. The weights' sum equals 1. R = W1\*R1 + W2\*R2 + W3\*R3 ...     - To determine if any votes diverge, boundaries can be set. To that end, delta values between each result are calculated and depending on the application maximum or minimum values are to be below a desired threshold. If there are any wild differences, error processing or recalculation can be initiated. Min/Max(Delta12, Delta23, Delta13) < Threshold     - There exist adaptive voting algorithms that determine correctness depending on previous values. They then pick the closest value to the expected one. * Timing Issues   + Care must be taken if elements are communicating over a network, and there exists significant latency. Obviously, there needs to be some sort of synchronization for correctly corresponding values to be compared to one another. * Routine Excercises   + Assuming there is an active and a stand-by element, we would like to trust the active element over the one standing by. In order to make sure that is the case, there needs to be a mechanism in place that monitors the active element's correctness. Routine Excercises check the active element in regular intervals. * Checksum   + If the results to be compared are very large, checksums can be used to reduce calculation time. * Losing votes   + Elements that lose the vote are to be treated as faulty and must begin error processing. |
| **System monitor** | 1. Problem   * The silent termination of operations in a fail-silent/crash failure mode is the best way for ensuring the system’s continued operation because it minimizes error propagation. However, when a part of system just stops silently, the other parts might not realize the failure and so they can not take the necessary steps for recovery. * The opposite of stopping silently is when the failing component is telling the world that it has a problem. By blindly reporting the problem to everyone, it uses up networking bandwidth and processing power. * An alternative is that the failing component just keeps ‘looking’ like it is working. This is a significant problem because it can fool everyone. This is a malicious failure and is very hard to detect.   2. Solution   * You could use Acknowledgement. They work well when the system can’t afford any messaging bandwidth or processing time to proactively monitor them. Downside of this approach is that it requires an ordinary exchange of information which can stimulate the acknowledgements. * Another solution is to create a mechanism that will actively monitor. Choosing a location for monitoring, it depends on the context. If the system is simple and it is executed on single processor, then monitor must be executed as separate task on same processor. If system is complex and hardware support is available, then monitor is a separate hardware. * The monitoring capability can be distributed throughout the system. Their activities must be coordinated and their responsibilities clearly defined. The monitor can monitor one component or several components, taking the correct actions when one of them stops. See Watchdog   3. Summary  Create a Monitor to study system behaviour, or the behaviour of specific parts of the system to make sure that they continue operating correctly. When the watched components stop, the monitor should report the occurrence to the FaultObserver and initiate corrective action. |
| **Acknowledgement** | 1. Problem  Normal operations consist of a request from one task to another and may or may not have a reply. In a client-server or peer-to-peer system there is a message flow that creates a two-way dialog between these tasks. Adding messages to the system increases the bandwidth requirements and can potentially delay more important information.  Given all that, what is the easiest way for one task to determine that the other task is alive and functioning?  2. Solution  Using mechanisms like heartbeat and watchdog adds a lot of complexity to the system, which in turn can lead to more faults. An easier way is to add acknowledging information to replies. This is referred to as piggybacking. This takes advantage of the fact that information about a failure is usually only needed when a task is needed.  Using the reply also comes with some disadvantages:   * If there aren't any requests then there will be no replies that can report a status (failure). If the system has the information before a task is needed, the failing task can be restored beforehand. * If a request takes a long time then the requestor might think that the request has died. An interim acknowledgement that merely acknowledges receipt of the request can help in these cases. * If the communication protocol does not require a reply then this mechanism obviously won't work.   Therefore, send acknowledgements for all requests. All requests should require a reply to acknowledge receipt and to indicate that the monitored system is alive and functioning. If no acknowledgement is received, then a failure should be reported to the fault observer and error processing should be initiated.  https://wiki.ifs.hsr.ch/APF/files/ack.png |
| **Heartbeat** | 1. Problem   * If the SystemMonitor does not have any indication that a task is still functioning it will be assumed to have failed and error processing will be taken. Unnecessary error processing wastes system resources. * If the activity level is low then the monitored task might look dead. * Sometimes the task being monitored does not realize that it is being watched. This occurs when an existing application is being integrated into a highly available environment. When the existing application gains new responsibilities, sometimes it cannot be altered, so it can’t be assume to be a more active part of the monitoring relationship.   2. Solution   * The health reports, or heartbeats, should occur at regular intervals. The timing enables the System Monitor to detect that the monitored task has fallen behind or missed a report. * In case of 3. problem (see above), System Monitor should initiate a request for a status report or ‘heartbeat’. * A ‘ping like’ message is a very simple way for the System Monitor. The message will stimulate an Acknowledgement response. However, too many of these messages, both in frequency and in number, can congest the messaging system. * Another way for the monitored task to report that it is still processing is for it to regularly execute a system call that was created for heart beating purposes. Failure to execute the system call at a predetermined interval indicates a failure.   3. Summary  The SystemMonitor should see a periodic heartbeat from the monitored task. If the monitored task does not supply a heartbeat response within the required time then recovery action should be taken. |
| **Watchdog** | 1. Problem  You want to have a system monitor (previous pattern) but you are worried about the added software complexity or the added overhead with the additional messages. Maybe you should rely on specialized hardware to implement the monitoring function. The question is: How can the system ensure a task is alive with a simple mechanism and you can't or don't want to add to messaging or processing overhead?  2. Solution  One way of monitoring without increasing the complexity is to watch activities that happen routinely. The monitor can ride on the coat tails of the monitored activities.  Another technique that adds only a little to the complexity of the system is to set a hardware time before critical operations. At the end of the operation the timer is checked to determine if the timing was within acceptable limits.  Sometimes you can add new hardware to the design. A simple component that will watch the monitored task's execution. It might monitor the control leads of a microcontroller, or it might watch a word in memory that the task is known to use, or it could be a passive observer watching an exchange of messages.  3. Watchdog  A hardware of software component depending on the system requirements, but in either case it will watch visible effects of the monitored task. The monitored task will not be modified. The watchdog should take some actions to get the monitored task back into the flock if it strays too far from expected and desired behaviour.  One way of implementing this is via peepholes or hardware test points to enable the watchdog to look inside the task  4. Difference with system monitor  Its very similar to the system monitor. The key difference is that the watchdog is assigned to watch only one task whereas the system monitor watches a number of tasks. |
| **Riding over transients** | 1. Problem  Not all faults are permanent. Some of them disappear after a time (take for example a lightning strike - it's gone as quickly as it came).  The system may ESCALATE error processing until the error is corrected - or masked. Many times the first signs of an error are not its true signature. Premature processing can result in taking incorrect actions.  How can the system avoid wasting resources processing transient (non-permanent) errors that won't have a long term effect on the system?  2. Solution  Not all errors need processing, because they disappear on their own. Conducting FAULT CORRELATION, errors can be categorized. Thus, the system can decide whether a transient error can be ignored or if it needs to be corrected. If it is only a transient error, keep monitoring it and take no action, unless it occurs more often than expected.  One lazy way to implement this is to not read the return code of disk writes. The writer assumes the disk is working correctly. If it had a non-transient error SOMEONE IN CHARGE would have already initiated error processing. Reading the return code might create a false alarm and unnecessarily start error processing.  The pattern LEAKY BUCKET COUNTER is one way to implement RIDING OVER TRANSIENTS. |
| **Leaky Bucket Counter** | 1. Problem  You are designing a system that is intended to recognize and correct errors automatically. The system needs to know whether a problem occurs regularly (intermittent), or just once (transient). This pattern can be applied to systems, where individual transactions are non-critical and can be ignored, or processed incorrectly.  How does the system know if an error is transient or intermittent?  2. Solution  Not all faults are permanent. There are some transient faults that are very hard to track down and reproduce, because most of the time they remain dormant, but under certain circumstances they rear their ugly head. Some examples for such transient faults are signal crosstalk, ESD (electrostatic discharge), lightning strikes etc.  Most of the time, errors are tolerable as long as they don't occur too often in quick succession.  That is where the Leaky Bucket comes in. A counter with a predetermined base value and upper threshold is initialized and every time an error occurs, it is incremented. At certain predetermined intervals, the counter is decremented independent of whether there was an error or not but never under the base value. If the counter exceeds the upper threshold, the system assumes that there is an intermittent problem and error processing is initiated.  Setting these three variables (base value, upper threshold and empty rate) are the system designer's task, since they strongly depend on the application. A real-life example is the 4ESS Switch - historically used in the telephone industry - that has a base value of 0 assumed, an upper threshold of 3 and decrements every 30 seconds. |
| **Realistic threshold** | 1. Problem  You are implementing a SYSTEM MONITOR or something with similar that monitors critical functionality.  How much time should elapse before the SYSTEM MONITOR takes action when an error is detected?  2. Solution  There are to different times to consider, when setting SYSTEM MONITOR thresholds:   * Message latency is the time between monitoring messages and should factor in the time it takes to send, receive and process any given monitoring message * Detection latency is the time after which the SYSTEM MONITOR should report a failure when no response from the monitored task is received   https://wiki.ifs.hsr.ch/APF/files/LatencyTerminology.JPG  Set the messaging latency based upon the worst case communications time combined with the time required to process one message on both ends.  Set the detection latency based upon the criticality of the functionality. Make it a multiple of the messaging latency. Use a smaller multiple for extremely critical or unique tasks, larger for REDUNDANT tasks.  Set the latencies so that the monitor will be informed in a timely manner to meet the availability requirement, but not too short that false triggers don't occur.  Shorter latencies can be implemented with the help of RIDING OVER TRANSIENTS and LEAKY BUCKET COUNTER.  The system may also be configured to dynamically adjust thresholds based on workload. |
| **Routing maintenance** | 1. Problem  How can we keep preventable errors from occurring?  Memory leaks are a type of fault that occur in many systems. In general, terms a memory leak is a loss of finite resources due to faults. Then these faults are detected early, the system can prevent errors and failures by not allowing the fault to activate.  2. Solution  Perform routine, preventive maintenance on the system.  A litt preventive maintenance avoids many common and easily preventable errors. Audits can be executed on a routine basis as can hardware exercises, (routine audits, routine, exercises) The preventive maintenance can be initiated either through an operator request through the (maintenance interface) or it can be built into the system to happen automatically. One way to do this is to use a low priority to the tasks, so the scheduler will execute it when the system isn't busy. |
| **Routine exercise** | 1. Problem  how do you know that redundant elements that will be called into service by a failover in case of an error or failure will actually work?  Failure free operations do not exercise all of the hardware of all of the software in a system. some of the hardware and software are intended for operation only when there has been a failure. Without periodic usage, these components do not have the opportunity to report problems with their use.  2. Solution  Checking for latent faults can be accomplished by using the standby element to execute the application. Make the standby become the active unit. If it is fault-free then it should safely handle the whole load. If it has latent errors then it will fail. An effective way to detect hardware transient faults is to perform the same execution at different times.  Generally, this pattern applies to hardware units, although standby software modules can be exercised also.  In general, execute the (routine exercises) during periods of low activity. Using this pattern technique must be decided on an individual basis. Not every component in the system is important enough to operation that it needs exercising.  https://wiki.ifs.hsr.ch/APF/files/routine_exercise_u11.PNG |
| **Routine audits** | 1. Problem   * Faulty data can exist in the system for a long time before it causes errors. Faulty data can easily propagate throughout the system. If the period between detection and recovery is long, then it's more likely that another task uses the same faulty data and it causes further errors. * The desired approach is to discover flawed data in a controlled way rather than to encounter during normal operations. This is very similar to fail-silent failure. Otherwise the data have been propagated to other parts in the meantime. * Checking all the data in a system for faults takes time. The system could check each data for errors by accessing it, which would disadvantage the performance. (similar to CompleteParameterChecking)   2. Solution   * The checking could also be done at a time when normal processing won’t be effected. This might be in an idle time slot, or it might be a maintenance task. A good performance engineer would schedule the processors so that they are not fully occupied and they have some capacity for RoutineMaintenance. * Many operating systems make use of a system idle process. Tasks can be run during idle time to perform RoutineMaintenance. In a priority scheduler the routine tasks have low priority. During the routine tasks, the system can access the data and determine its correctness. Any detection of data errors should be logged to allow for causal analysis, so the occurrences should be reported to the FaultObserver. * Where there’s one error in data there might be others. Correspondingly take cautious steps to consider what other data might be faulty.   3. Summary  Routinely check data to find latent faults. This checking should be done during idle times. If faults are found invoke U11\_3\_CorrectingAudits\_HS17 to correct and report the fault. |
| **Checksum** | 1. Problem  Checking whether data is correct can easily be done by maintaining a redundant copy of the data. But that becomes impractical when the amount of data to be checked is very large.  One way to work around this problem is to compute some aggregate value of the data being checked. For example, by calculating the sum of a structure that contains several integers. This will only work, if the ordering of the values doesn't matter, since reordering the values will still result in the same sum.  2. Solution  Checksums avoid the problem just mentioned. They are computed in a way that data values can be individually checked. Their purpose is to only check whether the data is correct. Going backwards from a checksum to the original data is not required (although possible if enough information is saved).  A simple example of a checksum is the parity bit. It serves as a flag that the data it is watching has changed. Parity only detects one error. Another example are hashing algorithms such as SHA or MD5 that are often used to compute a checksum over a block of information. They can be used to enhance fault tolerance by checking data that is received over transmission lines or stored on the system.  https://wiki.ifs.hsr.ch/APF/files/ParityBit1.PNG  It is important to keep in mind that when data watched by a checksum is altered during normal operation, the checksum must be recomputed. Periodically recomputing checksums for static data serves both as a validity audit and removes errors that might have been introduced into the checksum itself. |
| **Error Recovery** | |
| **Language Map** | https://wiki.ifs.hsr.ch/APF/files/recovery_patterns_u11.PNG |
| **Introduction** | Error detection does not fix the error. The error is still present in the system and can still cause a failure. The patterns in this chapter address error recovery, one of the two means of processing the error and resuming error-free operation. Error recovery continues execution even with a detected error by placing the system in a state that does not contain the error. the other means of error processing is error mitigation to mask the error. The next chapter discusses error mitigation.  Error recovery consists of two main parts.   * The first part involves undoing the bad effects of the error. * The second part involves creating an error free state in the system that can resume execution.   Both must use a minimum amount of time to maximize availability. As always redundancy is used when possible to produce the fastest possible recovery.  Many of these patterns involve preserving system state through checkpoints. A checkpoint is an incrementally saved state that facilitates rapidly restoring processing to a point at which the state was saved. This method allows for a high availability of the system.  However, restoring a saved checkpoint is only part of error recovery. Many of these patterns correct errors by changing the system state to one where the error does not exist, for example by resuming execution at the point before the fault activated. |
| **Overview** | ../../../../Downloads/IMG_B8E2AFBE0CE4-1.jpeg |
| **Checkpoint** | 1. Problem  You are designing a fault tolerant system that needs high availability (e.g. telecom) and/or relies on correctness (e.g. banking). The work requires significant processing time. An error may lead to work being lost which the system may take a lot of time to recover from.  How to prevent/minimize significant work and data loss and reduce recovery time?  2. Solution   * Saving system state incrementally and falling back to this checkpoint can significantly improve recovery time. * However, care must be taken not to save too often - which may lead to too big of a storage and work overhead - and neither too seldom, lest too much work might be lost and the checkpoint's usefulness may therefore be diminished. Balancing checkpoint frequency depends very much on the rate of failure. * For example, unmanned spacecraft make the equivalent of a checkpoint before they send data to earth. In case of a communication failure, data will not be lost. * An additional advantage manifests when using REDUNDANCY. Work may continue from a checkpoint on a redundant unit very quickly, reducing recovery time drastically. * A further concern is location: where is the checkpoint to be saved to? Certainly, not on the same UNIT OF MITIGATION, for obvious reasons. Nevertheless, it must be quickly available. * Recurring software problems pose another problem. Therefore, LIMITING RETRIES plays an important factor, otherwise the system may be stuck in an indefinite loop of error recovery. * Lastly, if INDIVIDUALS DECIDE TIMING, synchronization becomes an issue. The checkpoint must be done on all units at the same time, otherwise it becomes inconsistent.   3. Advantages   * Faster recovery from failure state * less data/work loss   4. Drawbacks & Pitfalls   * Requires significant work & data overhead * Requires the right checkpoint timing and synchronization among parts |
| **Restart** | 1. Problem   * The error is really bad. None of the other mechanisms to recover from the error are appropriate. Some errors require very drastic action. The SomeoneInCharge might have overseen a series of Escalation steps, but none of the steps resolved the error. This is a truly persistent fault. * Restarting the application can resolve the error if the fault is in the application. If the fault is in the underlying hardware, operating system or network then the application restart will not help.   2. Solution   * The restart is a pre-determined safe place for most applications and systems. Different levels of restart can be implemented that take different amounts of time. * A ‘cold’ restart level will initialize everything as it were the initial power-on of the system after a unexpected shutdown. It also includes extra checks which are executed to ensure that the file system is in the proper state. * A ‘warm’ restart can restore the system to an initial value but it skips some long steps. This is based on the assumption that the skipped steps are not needed because the file system is intact. * The restarts can be done at a very fine level. For example, restarting only small groups of Java containers in an application server environment. This technique is very effective against transient errors. If the restart does not succeed then restart larger groups is possible as Escalation.   3. Summary  Resume execution by restarting the program from the beginning. |
| **Rollback** | 1. Problem   * The system is in a context where accuracy is very important, and no bit of incoming information can be ignored. Requests, which were begun or arrived between the error occurrence and recovery completion, should have an attempt to be processed properly. Where should processing resume after error recovery?   2. Solution   * The system will need to be returned to a known place in its execution where the components are synchronized. The point, where a new work request is begun, is an example of a synchronization point. * If a CheckPoint is available then the system state can be restored to the one. If there are no checkpoints then the rollback should take processing back to right before the last requests were started. Requests should be saved until it is completed. So that if a rollback occurs the requests still exist and can be processed again. * When the system state is rolled back, some work will be done twice. Care must be taken to ensure that there are no side effects in repeated work. Redoing some of the work prevents rolling back from being a useful solution in some hard real-time applications, where deadlines might be missed when work is redone. * Depending upon what system state will be restored during the Rollback there is the chance that the error will reoccur. Limit Retries prevents repeating a cycle of errors and error recovery.   3. Summary  Return to a point where processing can be synchronized that is before the point of error. |

|  |  |
| --- | --- |
| **RollForward** | 1. Problem  After an error has occurred and error recovery has been completed, the system must determine where processing should resume.  Returning to the point of the error will most likely result in the same error again. Using a Checkpoint close to the time of the error also has a high risk of starting a cycle of errors and error processing, unless something has changed in the context. Another problem of performing a rollback to a point close to the error is that it can result in work being done twice or incompletely.  2. Solution  Instead of going back to a previous state, use Roll-Forward to move to a future state that the system would've encountered eventually if the error had not occurred. But before a Roll-Forward can be made, the damage caused by the error must be assessed to be certain that the point being advanced to will be valid. This is to prevent propagation and repetition of the error.  Many fault tolerant systems are event driven. This means that they are responding to external stimuli (e.g. incoming web service requests). In these kinds of systems rolling forwards causes the system to ignore the rest of the current stimuli, discarding it, and jumping to the point in execution where the next stimuli is received. Of course, that means that this won't work if the transaction cannot be ignored.  The decision to take this action should be made by Someone in Charge and the actions should be reported to the Fault Observer so that the outside world can be informed of the action. A System Monitor can watch the recovery action to make sure that it starts and finishes as planned. |
| **Return to Reference Point** | 1. Problem  Sometimes errors are so bad that neither Rollback nor Roll-Forward are the appropriate actions, but not quite so bad that Restart is needed.  This can be the case when an error is detected in an execution flow that is part of support processing (e.g. maintenance, error processing). Using Rollback or Roll-Forward in these cases only helps execution of the 'support code', but not the main application. The main application would benefit from returning directly to application processing. A Restart would solve the problem, but errors in support processing don't mean that the application needs to be restarted from the beginning, which takes time and affects the whole system.  2. Solution  Rollback points are dynamically created during execution and they change with time, which means they won't always be valid or useful as recovery points. Reference points, on the other hand, are static and because of that always available as recovery points.  Reference points are created at design time. One or more points are created for any given module. Some examples of good reference points are:   * where the task is entering its own initialization stage * where the task is exiting its own initialization stage * where the support task is about to return to the main application   In the C language 'longjmp' provides the mechanism to Return to a Reference Point.  The decision to take this action should be made by Someone in Charge and the actions should be reported to the Fault Observer so that the outside world can be informed of the action. A System Monitor can watch the recovery action to make sure that it starts and finishes as planned. |
| **Limit Retries** | 1. Problem  Faults are deterministic; when a latent fault is given the same stimuli, it will activate in the same way. Reprocessing stimuli from before an error can result in the error reoccurring. This is especially prevalent when using checkpoints.  How to prevent the system from being stuck in a recovery loop?  2. Solution   * When a 'killer' message (one that shuts down the unit) is stored in a checkpoint, it will repeatedly shutdown the component, even in case of redundancy. If single messages are not critical, one solution to break the loop is to flush the message queue (or more generally, the checkpoint). If the system keeps track on which message is being processed, it may even get away with just deleting only the problem-inducing message. * If messages can't be deleted, suspicious messages may be put into a special QUARANTINE queue. * If messages are not buffered internally, suspicious messages should be logged. Recurring killer messages could be a sign of a DOS attack. Limiting such attacks is crucial. * if the failure stems from an internal action, ROLLING FORWARD can prevent this event from being regenerated. That way it is not necessary to take down part of the system. * Other counter measures are RETURN TO REFERENCE POINT and RESTART, which might not re-simulate the fault.   3. Advantages  Prevent endless recovery loops  4. Drawbacks  Possible data loss depending on implementation |

|  |  |  |
| --- | --- | --- |
| **Error Mitigation Pattern** | | |
| **Language Map** |  | |
| **Introduction** | In the previous chapter error recovery patterns, we discussed how to escape from a failure state and bring the system back to a working state. In this chapter, we are going to talk about measures that don't affect system state, meaning that the system can continue working while error processing is going on. The techniques used mask errors and compensate for their effects.  Generally, there are two manifestations of errors: either an incorrect system state (incl. wrong value), or a missed timing requirement. If the error is of a faulty state kind, instead of bringing the system to an errorless state, the error or fault is corrected directly. In case of a timing related error, the system is given the means to power through until it returns to a normal state. The underlying fault may not be directly addressed because it might not be local to the overloaded system.  In the first chapter, we discussed that most fault tolerant systems are designed to only handle 1 concurrent fault. This chapter discusses means to handle multiple ones. Performance related mitigation techniques don't generally block other error detection and processing activities because they don't change system state.  If the system is handling a lot of work, even when technically overloaded, it must be working well and it is likely to be fine. But this can change rapidly, thus it is important to REASSESS OVERLOAD DECISIONS periodically.  Automatic actions related to timing errors fall into two categories:   * EXPANSIVE AUTOMATIC CONTROLS expand resources for the system, e.g. allocate more memory or processing power when the system is falling behind * PROTECTIVE AUTOMATIC CONTROLS protect the already limited resources from failing or being compromised. Firewalls and workload barriers are such examples   Another question when dealing with overloaded systems is whether to concentrate on the already started work (FINISH WORK IN PROGRESS), or prioritize new incoming work (FRESH WORK BEFORE STALE). Generally, it is more useful to do the latter if the requests come from people or other systems.  Faulty data can be marked (i.e. with a flag) so other parts of the system know not to rely on it, this prevents error propagation. If there is enough redundant information, the data may even be restored (ERROR CORRECTING CODES). | |
| **Overload Toolboxes** | 1. Problem  The system uses fault correlation and has determined that an error is not caused by faulty hardware or software, but by an excess of workload. This might be due to internal problems like memory leaks, or maintenance work (which should really be handled through fault recovery system), or it may be due to too many incoming requests.  How should the system handle situations of overload?  2. Solution  In an overloaded state, the system should handle as many requests as possible and degrade as little as possible. There are multiple factors to consider:   * Memory: the system needs more memory to store requests (e.g. if requests tend to peak, not very useful for constant load) * Tangible resources: requests require peripheral resources like printers that can't handle the workload * CPU load: requests require a lot of processing time and the system can't handle it.   These symptoms require different counter-measures and can't be dealt with in the same way. For example, if the requests tend to be very peaky, but aren't highly time critical but need processing, increasing memory to store requests and process them as system resources allow is an option (QUEUE FOR RESOURCES). Alternatively, FRESH WORK BEFORE STALE can also be an answer under the right circumstances to SHED LOAD.  When using redundancy, an overloaded peer may notify its neighbours that it is overloaded SHARE THE LOAD (a typical application of that would be load balancers from VSS). | |
| **Deferrable Work** | 1. Initial Situation  More new work is arriving than the system can normally handle, meaning it is in overload. Apart from the normal work there are also Routine Audits and Routine Maintenance tasks scheduled, to keep the system working well. But even though the workload is presenting a performance error / overload to the system, the system is stable. Workload is being processed at a very high level and the system will be able to ramp back down once the overload is over. In other words it isn't broken.  2. Problem  If a system is in this kind of overload state, there are two choices:   * Try to handle most of the new incoming work and defer the maintenance tasks. * Still perform the maintenance tasks at the expense of the normal workload.   3. Solution  Choosing number 1 is usually the correct choice in this situation because of a very simple reason:  https://wiki.ifs.hsr.ch/APF/files/DeferrableWork.png  Since the system is stable and using its resources to process the workload, it makes sense to defer the tasks that are needed when the system is not working properly, such as Routine Maintenance, Routine Audits. If the system is not broken and merely working hard, then it doesn't need to be fixed.  But although it is ok to defer some tasks that are not directly related to the primary purpose of the system, they should never be deferred forever. Because that lowers the overall fault tolerance of the system.  4. Exception  In case the system just seems like it's in overload, but is actually reacting to errors, Someone in Charge should employ Reassess Overload Decision. Reassess Overload Decision is also useful when this strategy is not providing enough relief and the system is not really stable. | |
| **Reassess Overload Decision** | 1. Problem  The system is overloaded with work requests and already employing error mitigation techniques such as Deferrable Work to reduce the impact of the overload. The decisions about what techniques to employ were based on Fault Correlation. But the workload mitigation techniques being used are not working to diminish the workload.  2. Solution  If the mitigation techniques are not enough to bring the system out of saturation, then the error may be coming from a source other than traffic overload. Provide the system with a feedback loop which provides information to enable the system to re-examine Fault Correlation decisions.  https://wiki.ifs.hsr.ch/APF/files/FeedbackLoop.png  This enables the system to attempt different error processing strategies that might help solve the problem. | |
| **Equitable Resource Allocation** | 1. Problem   * How should requests for scarce resources be handled? * The arriving requests might arrive differently in either type or priority. An example for type, many requests are just for visiting the online shop. But only few requests are placing an order. An example of different priority, salesmen can access company's server with higher priority than ordinary user. * The system could strictly follow FreshWorkBeforeStale (LIFO concept) and only give the newest requests service. This will result that request with high priority will not be processed because the newly arriving low-priority requests are always processed. * If different types of requests are routed to different parts of system for processing, then the part with less common requests will have some idle time and the part with more common request will have more work. * There might be a specific resource which is especially overloaded. To sum up, if requests are allocated based only upon their newness or other preferences, they might end up blocking this resource anyway.   2. Solution   * The strategy would be to look at all the requests for service, both fresh and queued, and allocate resources equitably (= gerecht) to all of them. Downside is that it requires additional bookkeeping of request and their resource needs. But work can be redirected around specific resource overloads. This helps to provide service to as many requests as possible. * Priority inversion is a problem that results when a task of a lower priority is preventing a task of higher priority. Because the lower priority tasks hold a resource that is needed by the higher priority task. When distributing messages to queues, it is important to avoid creating the priority inversion-like situation.   3. Summary  Pool all similar requests and allocate resources to the pools based upon their availability and priority. This enables all types of work to be accomplished even if concentrated overloads exist. | |
| **Queue for Resources** | 1. Problem   * What should be done with requests for resources that cannot be handled immediately when they arrive? * An option is to discard all requests that it can’t handle immediately. Only those requests that can be handled will be kept. There are many flaws with this approach. First flaw is, when a big request is divided into multiple requests. And if the last request is discarded, then the big request cannot be completed. A further flaw is that important single work items might be eliminated without any consideration. In addition, the overload might be momentary and the requests could be processed later.   2. Solution   * The system stores the work in a queue for later. Such that a big request, which is split in multiple requests, can be completed eventually. * There are still risks with this approach. One is that the queue might become too long so it cannot be effectively managed. Another is that even after a short period in a queue the resources needed for a request are still not available, requiring the request to be rejected at that point, or put back in the queue. * Managing the queue requires resources and introduces overhead. You must be careful to not make the situation even worse with bad design. * By computer generated requests use FIFO queue because it must be processed in order. By human generated requests use LIFO queue because it will look like if your service good response time.   3. Summary  Store requests for service that cannot be handled immediately in a queue. Give the queue a finite length to improve the likelihood that the request is still important when it reaches the head of the line. | |
| **Marked Data** | 1. Problem  When the system detects an error, there are many ways to deal with it. If the error has a limited scope that does not require the system state to be altered greatly, but there is not enough information to immediately correct the error, then error mitigation is often more appropriate than error recovery.  2. Solution  In these cases, one way of dealing with the error is to mark it. Since we don't want that the erroneous data is used by any other parts of the system, the component that first detected the error should mark it in a way that other parts of the system do not have to spend much time detecting that it was erroneous.  An example for marking a value as erroneous is the IEEE 'Not a Number' (NaN). This value is stored in place of a floating-point value as the result of certain illegal floating-point operations (e.g. division by zero). The standard defines rules for how subsequent computations should behave when one of the operands is NaN. These rules should include two different types of information, which are both present in the NaN rules from the IEEE:   1. How should the operation proceed?  * Use a default value * Skip the operation and mark the result as erroneous * Seek the information from an alternate source * Abort the execution * Invoke an error handler  1. Should other parts of the system be notified when an erroneous value is encountered?  * Yes: Referred to as 'signaled' in the IEEE standard. * No: Referred to as 'quiet' in the IEEE standard   The signalling is appropriate if some intermediate mechanism would have been expected to correct the error, meaning the occurrence of the erroneous flag is unexpected.  Marking data is not free! In the case of IEEE NaN the mark is encoded in place of the value, but sometimes the mark might require additional 'meta-memory'. Resources to check for the mark and take appropriate actions are also required.  3. In Short  Mark erroneous data to indicate that it is unusable and define rules for processing these values when they are encountered. | |
| **Error Correcting Codes** | 1. Problem   * How can the system data paths and storage be made as error free as possible? * CheckSum can detect that an item has been corrupted. But it cannot correct the data automatically. * To reduce errors, the data should be corrected automatically. There are numerous techniques like RoutineAudits, Data Reset and CheckPoint. If a duplicate of data is maintained, then there is no need to correct single value.   2. Solution   * The mentioned techniques all work because they keep a copy of data. An alternative way is to undo the error by ignoring the value totally. It requires that a system knows exactly which part of the data has changed. Then the system can undo the change without knowing the value. The checksum tells only that data has changed, but it doesn't provide enough information. * If you use parity over smaller and smaller parts of data, you will eventually reach the point where system can tell what has changed. If one parity bit watches over one data bit, then system can know which data bit has changed. * To tell if the parity bit has changed, put another parity bit watching over it. Error correcting codes are a very fast way of ensuring correct data. This is especially valuable when there is a high rate of transient errors. Error correcting codes are frequently applied to data streams to ensure that they are received correctly. (e.g. MPEG data streams)   3. Summary  With each checksum store enough information that will enable the system to automatically correct a flawed data value when it is detected. | |
| **Expansive Automatic Control** | 1. Problem  The system can apply automatic controls when it becomes overloaded. The metric being watched is the number of work requests that are processed completely (view focus on incoming work).  How can we avoid both the wasted effort processing the requests that can't immediately be handled in an overload and at the same time increase overall request completions?  2. Description  This seems impossible. not waste effort queuing or discarding excess requests and increase the number of requests that can be handled. The resource that is exhausted is finite. Unless it can be increased this seems impossible.  If some way to expand the range of possibilities available to a system were possible then more work can be done.  3. Solution   * High performance systems are usually designed to operate at less than 100% processor occupancy during ordinary execution. The unallocated processor time is held in reserve to support occasional spikes or periodic tasks. It can also be released for general requests when the system is overloaded. * Another example of expanding the resources is to use another route, which may be longer but also can handle the connection. * Design some resources into the system that will be used only in case of overload. Provide new ways for the system to do its work that either uses reserved resources or fewer resources.   https://wiki.ifs.hsr.ch/APF/files/offered_load_u11.PNG   * Impose automatic controls that provide new ways to do the same work. This enables taking on excess work and lessening the congestion. It is like providing an escape valve.   https://wiki.ifs.hsr.ch/APF/files/additional_ways_u11.PNG  4. Practical example   * An example of an expansive control from highways is when the police direct traffic over side streets, possibly in violation of posted regulations, to avoid an accident scene. * Another example is using all lanes of a highway to carry traffic in one direction to allow a higher number of people to leave an area threatened by a natural disaster, such as a hurricane.   5. When to use which (expansive/protective)  This pattern applies when the resource is expandable. If there are no ways of expanding the range of possibilities - of increasing the amount of the resource - then apply protective automatic controls. | |
| **Protective Automatic Control** | 1. Problem   * The system can process errors automatically. It observes its throughput, that is the amount of requests processed per timeframe. The system has finite resources that may not be extended. * When resources are running close to capacity, the overhead that finds available resources rises as well to find the last free spots. These two factors combined pose a threat to the system. The overhead might stall out processing of the actual workload. * What actions should an overloaded system take to avoid spending all its time doing overhead work associated with new requests arriving?   2. Solution   * If the system needs to process all requests, they may QUEUED FOR RESOURCES until the resources are available. This requires additional overhead to queue and dequeue the messages. * When old messages turn obsolete quickly, eg. in services for humans, FRESH WORK BEFORE STALE can drop old requests in favour of new ones. * There are two effective strategies how to deal with an onslaught of requests. First off, the system may protect its resources zealously, in extreme cases even stop processing requests. * The second option is to open the floodgates and take on as many requests as possible and shedding internal processes (see DEFERRABLE WORK). Once necessary, it may SHED LOAD, ideally SHED LOAD AT PERIPHERY (before the system starts processing).   https://wiki.ifs.hsr.ch/APF/files/This-NotThis.PNG  Applying one of these to strategies allows the system to continue to perform, albeit with a hit to performance. However, if no action is taken, the system's performance will decrease drastically (see Figure 72). | |
| **Shed Load** | 1. Problem  The workload increases as requests are accepted for processing. If too many requests are accepted, the system throughput can decrease to the point that no work is being completed. When this happens, the system is unavailable.  How can the system best handle too many requests and keep them from overwhelming the system?  2. Solution  One way to solve these situations is by shedding load. Shedding load means not accepting and discarding some work.  This will enable the system to stabilize and process the accepted workload well. The work should be shed as early as possible, before it consumes many system resources, see Shed Work at Periphery.  The mechanism to shed work should consider how the communicating system will behave when its work requests are rejected. When the work is arriving in the form of messages, work can be shed by failing to acknowledge work request messages. Often the sender will retransmit the message. This is both good and bad. Good if when the retransmission arrives the overload is passed and the system can process it. Bad otherwise. | |
| **Final handling** | 1. Problem   * When requests finish, the resources that they used are routinely released back to process other requests. Sometimes a request terminates exceptionally before the point at which resources are released. These resources need be released. * Should there be a separate mechanism to support only the abnormally terminated request’s need to release resources? * The first option is to not clean up abnormally terminated transactions. This will leave resources tied up and unavailable for other requests. Like by memory leak the system will eventually run out of resources. The resources can be collected automatically by a garbage collection mechanism, but GC mechanisms doesn't bring the desired deterministic performance (at time the book was written. It not true anymore because e.g. language GO uses explicitly GC). * The easiest option is to release two separate mechanism. This simplifies the design stage because the normal and abnormal control flows do not need to interact, which reduces complexity. One mechanism for the normal cases and another for the abnormal cases. Separate mechanisms can be kept simple. This presents a maintenance challenge because now there are two different mechanisms that need to be maintained. If a latent fault is later identified in one of the mechanisms, all of them need to be checked to make certain that they do not have the same fault. Although the mechanisms are kept simple, the amount of code is doubled. With more code, there will be more latent faults.   2. Solution   * Another option is to design so that the normal and abnormal cases share the same resource release mechanism. This can result in a more complicated design since the control paths for normal/abnormal must be coordinated. This complexity opens the potential for faults in the design. But the resulting amount of code is only half as much compared to separate mechanisms. An advantage is that there is only one place to correct in case a fault is found. * Most fault tolerant systems are in use for a long time. As a result the costs to maintain the system must be considered in addition to the development costs. This means that for fault tolerant systems the shared mechanism is more suitable because the maintenance cost is lower than development cost by shared mechanism.   https://wiki.ifs.hsr.ch/APF/files/Screenshot_from_2017-12-07_07-15-33.png  3. Summary  (Like java finally block)  Integrate the release of resource for internally terminated transactions with the usual release of resources done by normal task termination. | |
| **Slow it down** | 1. Problem  A system that processes incoming messages can be overloaded to the point where it will not process anything anymore. Below this threshold, functionality degradation is gradual. The system can't wait for human intervention and uses MINIMIZE HUMAN INTERVENTION.  What should the system do when it has more requests for service than it can possibly handle efficiently?  2. Solution  The system should use ROUTINE MAINTENANCE, ROUTINE AUDITS, and a fault tolerant design to keep it healthy. Otherwise it won't be able to tell apart an error from excess load. It uses DEFERRABLE WORK to delay less critical maintenance tasks.  One method to deal with overload is to ignore it. Since this will lead to an eventual shutdown, it requires appropriate fault handlers like an overseer (human or not) to reboot.  By gradually reducing the work the system does per incoming request, the system can delay the point where it is overloaded.  The degradation should be performed in predefined ESCALATION steps instead of continuous adjustment, since this is easier to design and maintain. Of course, adding this complexity should not be done blindly and the benefits should always be weighed against the cost in complexity and difficulty in understanding the mechanism. | |
| Each step should restrict request processing more and more  https://wiki.ifs.hsr.ch/APF/files/SteppedRestrictions.JPG | To prevent oscillation between steps, a hysteresis effect should be added  https://wiki.ifs.hsr.ch/APF/files/Hysteresis.JPG |
| Let's look at a real-world example of SLOW IT DOWN, the 4ESS switch:  It uses 5 escalation steps to deal with system overload.  Level 0: the normal functioning state with no restrictions at all.  Level 1: Some maintenance and audit tasks are deferred. They are still allowed to work, but not as much  Level 2: Resources are taken from maintenance and audit. Blocking calls may be avoided like this.  Level 3: The system may begin to SHED LOAD. Critical 'call registers' are restricted. Some requests are ignored. The surrounding switches may redirect the requests elsewhere. There are further restrictions on maintenance.  Level 4: Only one third of the call registers may be allocated. Even more requests are ignored.  Level 5: The system stops accepting any new requests. Internal QUEUE FOR RESOURCES may be processed slowly. | |
| **Shed Work at periphery** | 1. Problem  If too many requests are arriving, the system can Shed Load. But the mechanism to shed work also consumes some system resources.  How does the system Shed Load for the lowest additional effort?  2. Solution  As more and more effort is expended completing a particular request, the benefits to the system of shedding that request diminish. Or in other words: Shed the request as early as possible, before it consumes any system resources.  Systems that accept network requests need some kind of peripheral equipment making them into a network element (which is where the name of this pattern comes from). This network component usually pre-processes incoming requests, meaning that it able to detect which work is eligible for shedding. Work shedding done as soon as it hits the boundary of the system (the network component) keeps the core of the system (which is usually the performance bottleneck) from seeing excess requests. | |
| **Finish work in progress** | 1. Situation  The work requests to the system are related to each other. Either each request builds upon earlier requests, or the requests might in some way alter the way in which some other request is handled. More requests than can be handled are arriving. The system needs to shed load and stop processing some requests.  2. Problem  Which requests should the system accept for processing and which should it reject? You do not want the system to reject a request that is the last piece needed to complete some super-request task that has been processing for a long time.  3. Solution  Process the requests that are continuations of work in progress. Ignore and reject the requests for new work. To do this effectively requires an inspection of incoming requests and the determination that the system is already processing other requests related to this new one.  There is a case where this strategy results in the system oscillating between overload and idle.  https://wiki.ifs.hsr.ch/APF/files/finish_work_cycle_u11.PNG  This occurs when admission of new tasks to the system is suspended for a period of time. When no more requests are admitted to the system, a drop in throughput occurs when the work in progress starts to complete because no work has been admitted to the system. After the work in progress completes new work is admitted, which, if still an overloading amount, will cause the cycle to repeat. | |
| **Fresh work before stale** | 1. Situation  More requests for service are arriving than the system can possibly handle. The requestors can abandon requests prior to completion by their requestors. The system can sort incoming requests into different categories, which enables it to finish work in progress and to intelligently shed load.  2. Problem  How can you ensure that the maximum number of requests get good service? When requests take a very long time to process, the requestors may give up waiting.  When the system is processing as many requests as it can, it will almost always need to queue requests before they can be served. The simplest way to queue, or buffer, the requests is to use a buffer that acts like a First In First Out queue. A problem with this is that requests that are in the queue might be abandoned by their requestor.  3. Solution  To ensure that some requests receive service quickly, process a few requests as soon as they arrive without any buffering. This ensures that the quality of service for at least a few requests will be excellent. Therefore, use a *Last In First Out* queue and also serve some new requests immediately. In that way, at least some requests will have a high quality of service.  If requests are related to each other then finish work in progress should be used to continue processing the work that has already been started. | |
| **Treatment Pattern** | | |
| **Language Map** | https://wiki.ifs.hsr.ch/APF/files/treatment_u11.PNG | |
| **Introduction** | After an error is processed, fault treatment prevents the error from reoccurring. This includes repairing the fault. This can mean patching the system, which is installing a new fault-free version of the software or the data. It can also mean correcting a procedure so that a person won't cause the same error again. The steps of fault removal are:   * Verification, to determine if the system behaviour conforms to its specifications. This is done to check whether the fault is still present in the system. * Diagnosis, determine the cause, both in terms of location of the fault and the nature of the fault * correction, fault is made passive. After that verification is again needed.   This is an example of corrective maintenance. | |
| **Let sleeping dogs lie** | 1. Problem  The system detected and corrected an error or failure. 'Fixing' a fault can lead to other faults. Should you correct all faults that the system and the maintainers find?  2. Solution  When deciding whether to fix a fault, one must weigh the benefits of the fix against the unknown potential faults introduced. Is the fix complex to do or does it add a lot of complexity?  The status quo is predictable, if changes are made that is no longer the case.  Furthermore, fixing a fault incurs costs. Not only development time, but also system down-time while the patch is applied. Is this hit to availability acceptable?  If the fault causes system instability and/or failures that result in unavailability, the fix might be appropriate. On the other hand, if there is a simple workaround, fixing the problem might be too much and it might be rolled in when a more dangerous fault is found and fixed. | |
| **Reproducible error** | 1. Problem  The system detected an error or failure and you are now interested in correcting it. Through the information that the system captured about the nature of the error or failure, you identified (or at least think that you have) the stimuli that triggers the fault to activate.  But the system has not been static in the time between when the error processing completed and you begin treating the fault. Faults are sometimes corrected and removed from the system as a side effect of some other fault treatment or Software Update.  Unless you very clearly identified the fault that caused the previously detected error, you cannot be certain that it is still present (unless it causes the error again). You need to be sure that it is really broken before you can fix it and that you are fixing the fault the causes the error or failure of interest.  2. Solution  Stimulate the fault in a controlled manner to verify that the fault did indeed cause the observed error and that the fault is still present in the system.  The system can be designed to automatically reapply the stimuli that has caused the error. Although this is not usually done because of the complexity that gets added to the system (more complexity -> more faults).  Automatic stimulation to detect latent faults is done in some systems as a form of system diagnostics. A Quarantine zone is established and sequences of stimuli are introduced to see if the system behaves according to specification. These diagnostic tests are very effective if run in conjunction with Routine Exercises.  The stimuli can also be introduced manually by the operating personnel.  When a fault activates, there's always a risk that the recovery will be unsuccessful and a failure will result. That's why the benefits of correcting the fault must be weighed with the benefits of leaving the fault alone (Let Sleeping Dogs Lie). If it is not corrected you should consider providing better error processing for this particular fault (or class of faults) in a future Software Update. | |
| **Small Patches** | 1. Problem   * You know how to correct the fault that caused a detected error or failure. You have weighed the options with LetSleepingDogsLie. You want to deploy correction with minimal risk of downtime. * What kind of update will have the least chance of introducing extra faults or bringing in extra capabilities that are not needed? * The likelihood of a fault increases as size of the update increases. With more code, there are more places for the fault to hide and more complexity to understand and test.   2. Solution   * Some tools allow you to create byte level difference records. These differences can be examined to determine if anything unexpected has changed, which is a clue that there are other faults within the update. An example is when you expect two to four bytes to be changed but instead the tools inform you that 3000 bytes changed. * The tools to install small, possibly bit or byte level, corrections in a system could be very complicated. Typically, they will require a portion be a permanent part of the system. Application of the correction might be quick, from downtime viewpoint. But the overall time that the system spends managing the correction can be significant. * In many cases the only way to correct a faulty computer program is to stop its execution, install the new version and restart it. It results a long period of unavailability because reinstalling the complete program or even updating to a new version is time–consuming. It also opens the system to corruption or installation of faults in other parts of the system. That's why care must be taken to ensure that the entire program update includes only the desired correction and doesn’t include anything else. * Finally, it depends on your tools. If they support the application of byte level updates then that method becomes an option. If the smallest change that you can make is to change an executable file, that defines the minimal size of the update.   3. Summary  Create as small an update as your tools and what needs to be patched allow. By reducing the size of changes to the system, you have reduced the risk of introducing new faults into the system. | |
| **Root cause analysis** | 1. Problem  An error or failure occurred. The error has been processed and the system returned to service. You have decided to correct the fault.  What fault should you fix?  2. Solution  When fixing a fault, you do not just want to look at the symptoms. Find out the root cause of the error/failure.  In figure 98 we can see cascading failures. We initially only see the symptoms of failure C. But in reality, it is caused by failure B, which in turn is caused by failure A. And its fault may lie somewhere else entirely.  Always ask why:   * Why was the data record lost? - Because the transaction failed in the middle. * Why did the transaction fail in the middle? - Because it ran out of memory. * Why did it run out of memory? - Because there was no more memory available for allocation. * Why was there no more memory available? - Because the memory was inaccessible. * Why was the memory inaccessible? - Because its owning task had terminated without releasing it.   With this we can now make sure that all tasks release memory before exiting. Usually, asking 'Why..?' about 5 times gives enough conclusive evidence.  Of course, there is not always just one fault. It might be a faulty procedure, and a procedure carried out incorrectly by the maintainer. None should be ignored, but they should be recorded and pursued as well. | |
| **Revise Procedure** | 1. Problem   * The RootCauseAnalysis identified some faults that were related to the way that the operating personnel behaved. The system design might have been incomplete in its implementation of MinimizeHumanIntervention. * How can a repetition of errors caused by the operating personnel be avoided? * True experts know what to do, so when they intervene they rarely cause a failure to become more severe. Their intervention aids error processing and reduces unavailability. (see MaximizeHumanParticipation) * Many of the people who are maintaining your system are not true experts in its operation. They may not know how to correctly interpret the error messages. When confronted with ambiguous, misleading, or poorly worded instructions they might just start typing/clicking anything in an attempt to correct the error. They are truly dangerous and will cause many failures. The procedures for them to follow must be very clear and painstakingly researched.   2. Solution   * Some of the people who are not experts will read the procedures that you have developed to explain how to manage the system. In some cases, errors can still occur, because the procedures that the people followed were not correct, were not complete, or were unclear. * Scenarios, such as software update, system configuration changes, and system backups are executed frequently and also benefit from predetermined procedures. Availability suffers when the procedures cannot be followed by the operating personnel.   3. Summary  When operating personnel following the system’s predetermined procedures contribute to failure durations, revise the procedures to avoid repeating the same sequence of errors. | |

**Pattern-Kombinatorik**

|  |  |  |
| --- | --- | --- |
| Pattern | Kombinatorik | Begründung |
| Abstract Factory | *Factory Method* | Oft sind Methoden der Abstract Factory als Factory Methods implementiert |
| *Prototyp* | Manchmal sind Methoden der Abstract Factory als Prototyp implementiert. Abstract Factory might store a set of Prototypes from which to clone and return product objects. |
| *Singleton* | wird meist als Singleton implementiert |
| Adapter | *nicht kombinierbar* | |
| Bridge | *nicht kombinierbar* | |
| Builder | *Prototyp* | verwendet Prototyp für Konstruktion |
| Chain of Responsibility | *Composite* | Um die "Chain" zu definieren → Baumstruktur |
| Command | *Composite* | aufgebaut als Composits → ein Command ruft mehrere andere Commands auf **?** |
| *Null Object* | Damit keine Null tests für Commands durchgeführt warden müssen |
| *Memento* | Command kann als "undo" Funtkion ein Memento bentuzen |
| Composite | *Builder* | Builder kann verwendet werden um Composits zu erstellen |
| *Decorater* | Mit Decorater einem Composite zusätzliche Funktionalität hinzufügen |
| *Flyweight* | Mit Fylweight können Composits geshared werden |
| *Iterator* | Mit dem Iterator können Beziehungen von Composits durchlaufen werden |
| *Chain of responsibility* |  |
| *Visitor* | um Operationen hinzuzufügen → Blätter des Trees können mit Visitor bearbeitet werden |
| Decorater | *Strategy* | Chaning guts |
| Facade | *Singleton* | wird meist als Singleton implementiert |
| Interpreter | *Composite* | Nutzt Composite um seine Grammatik zu definieren |
| *Flyweight* | Um Symbole zu sharen |
| *Visitor* | Um Operationen hinzuzufügen |
| Iterator | *Memento* | Um den Zustand der Iteration zu speichern |
| *Null Object* | Um auch Iterator-Funktionen auf einer leeren «Null»-Liste aufrufen zu können |
| Observer | *Mediator* | Observer kann Mediator für komplexes Management von Dependencies nutzen |
| Proxy | *nicht kombinierbar* | |
| Singleton | *nicht kombinierbar → antipattern* | |
| State | *Fylweight* | Sharing State Objects |
| *Null Object* | Kein aktiver State |
| Strategy | *Decorater* | Changing skin |
| *Flyweight* | Um Strategien zu sharen |
| *Null Object* | Standard / untätig-Strategy |
| Template Method | *Factory Method* | TM nutzt oft FM für seine Hook Methoden |
| *Strategy* | Um die Schritte des Algorithmus zu definieren |
| Visitor | *Iterator* | Defining traversals |

Unterschied Adapter vs. Decorater

- Adapter passt Schnittstelle an und hat Verhalten identisch

- Decorator haben gleiche Schnittstelle und passt verhalten an

 