School of Engineering Science
Software Engineering
CT10A7004 - Sustainability and IT
Prof. Jari Porras

Task II

Exploration of Gang of Four Design Patterns

Monday, 13th March 2023

Jan-Nicklas Adler - 001466753
Cindy Aprilia - 001185063
Karolina Bargiel - 001186266
Elmar Karimov - 001185186
Apoorva Nalini Pradeep Kumar - 001185856
Martin Mayr - 001467244
Rosheen Naeem - 001185649
Korawit Rupanya - 001187113

Contents

1. **Pattern Name: Decorator**

Common uses:
In object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class

Energy saving rationale:
As the decorator pattern simplifies code its energy efficiency would be better. Decorator allows for code improvements even after the classes are created, which also allows for higher maintainability of the code. In the following study, the researchers found an increase in energy efficiency between 4 and 25 % for the Decorator pattern.

When it should be used:
The decorator pattern is commonly used when:

- When an object requires the extension ex. a window control that requires additional "optional" features like scrollbars, title bar, and status bar.
- When Several objects support the extension by "decoration". Usually, those objects share a common interface, traits, or superclass, and sometimes, additional, intermediate super-classes.
- The decorated object (class or prototype instantiation), and the decorator objects have one or several common features. In order to ensure that functionality, the decorated object & the decorators have a common interface, traits, or class inheritance.

When it should not be used:
A decorator should not be used in cases where the additional features are so complex the whole class should be rewritten.

2. **Pattern Name:** Facade

Common uses :
The facade pattern in which an object serves as a front-facing interface masking more complex underlying or structural code.

This pattern is commonly used when:

- a simple interface is required to access a complex system,
- a system is very complex or difficult to understand,
- an entry point is needed to each level of layered software, or
- the abstractions and implementations of a subsystem are tightly coupled.

Energy saving rationale:
Thanks to the implementation of a simple interface the energy usage on the client side is reduced dramatically. Additionally, the pattern supports loose coupling which in general has been proven to reduce energy consumption;.

When it should be used:
A facade can:

- improve the readability and usability of a software library by masking interaction with more complex components behind a single (and often simplified) API
- provide a context-specific interface to more generic functionality (complete with context-specific input validation)
- serve as a launching point for a broader refactor of monolithic or tightly-coupled systems in favour of more loosely-coupled code

When it should not be used:
The alternative approach is called dependency injection. Some programmers do advise using that over the facade patterns because it allows for a better object-oriented design and better usage for design patterns when there’s complex code overall.

3. **Pattern Name:** Factory method

Common uses:

Factory Method is used to make objects without specifying their concrete classes. Instead, by defining an interface for object creation, the Factory Method lets subclasses choose which class to create.

The Factory Method design is most frequently used when one of several subclasses in a class hierarchy needs to be instantiated based on some criteria or input.

Consider a scenario in which your application needs to produce various kinds of documents, including text, spreadsheets, and PDFs. You could build subclasses for each type of document by first creating a DocumentFactory interface with a method for creating a new document using the Factory Method pattern. Without needing to know the specific implementation class, the client code can then call the createDocument method on the factory to create the desired document type.

Energy saving rationale:

Although this pattern doesn't directly relate to energy savings, it can nonetheless tangentially support energy efficiency by enhancing software system performance and lowering resource usage. The client code can assign the duty of object creation to a different class or module by using the factory method design. This not only makes the client code simpler but also makes the codebase's structure and maintainability better. The software system improves in efficiency and becomes less prone to mistakes and bugs as a consequence.

Efficient software systems require less computational resources to execute, which translates to lower energy consumption. Additionally, well-organized and maintainable codebases can be optimized for performance and efficiency, further reducing energy consumption.

The Factory Method design also encourages code reuse by enabling the abstraction of object creation from client code. This allows different system components to use the same object creation logic, decreasing duplication and conserving computational resources.

When it should be used:

The Factory Method pattern should be used in situations where you need to create objects without specifying their concrete classes. The pattern is particularly useful in the following scenarios:

1. When you have a class hierarchy with multiple subclasses, and the client code needs to create objects from one of those subclasses dynamically based on some criteria or input.
2. When you want to decouple object creation from the client code and provide a way to create objects based on some input or criteria.
3. When you want to promote code reuse by allowing multiple parts of the system to share the same object creation logic.
4. When you want to simplify the client code by delegating the responsibility of object creation to a separate class or module.

When it should not be used:

While the Factory Method pattern can be useful in many scenarios, there are also situations where it may not be the best choice. Here are some cases where the Factory Method pattern should not be used:

1. When there are only a few concrete classes in the class hierarchy, and the creation of objects can be easily handled by the client code without creating a separate class for object creation.
2. When the creation of objects is simple and does not require any complex object creation logic. In such cases, using the Factory Method pattern may introduce unnecessary complexity and overhead.
3. When the system needs to create objects frequently, and the overhead of creating a new factory object each time can impact performance. In such cases, other creational patterns such as Singleton or Object Pool may be more appropriate.
4. When the object creation process requires a large amount of configuration or data that is difficult to pass through the Factory Method interface. In such cases, a different pattern such as Abstract Factory or Builder may be more suitable.

4. **Pattern Name:** Adapter

Common uses :
The Adapter Pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate

This pattern is commonly used when:

- Interfaces of an application can not communicate with each other due to structural differences of the data.

Energy saving rationale:
The Adapter pattern can be used to optimize the energy consumption of a system by making objects interconactable allowing swapping out components of the software. This can reduce required maintenance and lower the energy cost required for the developement/maintainance of the system.

When it should be used:

- An existing class or interface that cannot be modified is present in the codebase, but needs to be used with another class or interface that has a different interface.
- A more flexible system should be created by allowing different classes or interfaces to work together interchangeably.
- A third-party library or service that has a different interface than the rest of the codebase should be integrated.

When it should not be used:

- When the interfaces of the two classes are very different and cannot be easily mapped to one another.
- When performance is a critical concern, adding an adapter layer can introduce additional overhead and reduce system performance.
- When the interfaces are expected to change frequently, creating adapters for each change can result in a large number of adapter classes, making the system more complex and harder to maintain

5. **Pattern Name:** Iterator

Common uses :
Iterator is a behavioral design pattern that allows to traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

This pattern is commonly used when:

- Support for traversal of a collection while modifying it at the same time, using the same iterator object is needed.
- When a default way to iterate over different types of collections, such as arrays, linked lists, trees, and graphs should be available.

Energy saving rationale:
The Iterator pattern does not help to save energy per se but can help to make a system more performant and therefore saving energy. I.e. Lazy Evaluation of objects or the fact thaht list can be traversed and changed at the same time can avoid extra iterations over the datasets.

When it should be used:

- An existing class or interface that cannot be modified is present in the codebase, but needs to be used with another class or interface that has a different interface.
- A more flexible system should be created by allowing different classes or interfaces to work together interchangeably.
- A third-party library or service that has a different interface than the rest of the codebase should be integrated.

When it should not be used:

- When iteration and change of the date is not required in the same step
- When the dataset is small then more more intuitive ways of iterating the data might be better

6. **Pattern Name:** Observer

Common uses:

The Observer pattern is a behavioral design pattern that you use when you need to keep track of a list of objects that rely on the state of one object and update them when that object's state changes. The subject, or object being observed, and the observers, or objects depending on the state of the subject, are the two major parts of the observer pattern.

The Observer pattern is most typically used in user interface programming, where updates to one UI component may cause changes to other UI components. In a spreadsheet application, for instance, other cells that rely on a user-updated cell's value should also be updated automatically. The subject is the cell that is being watched, and the observers are the cells that rely on its value.

Energy saving rationale:

While this pattern does not directly relate to energy savings, it can indirectly contribute to energy efficiency by reducing the computational resources required to handle updates in software systems.

Without the Observer design, dependent objects would have to poll the subject object frequently to check for state changes. This could lead to unnecessary energy consumption and call for a substantial amount of computational resources.

Continuous polling can be avoided by using the Observer design, which notifies dependent objects when the subject object's state changes. This strategy is more effective and can lower the amount of energy used by software systems.

The Observer pattern also encourages loose coupling between objects, which can boost software systems' general effectiveness and performance. Better worry separation is made possible by loose coupling, which facilitates performance and efficiency optimization of the system's individual components.

When it should be used:

The Observer pattern should be used in situations where you have a one-to-many relationship between objects, and you need to notify the dependent objects when a change occurs in the state of the subject object. Here are some specific scenarios where the Observer pattern can be useful:

1. When you need to decouple the subject object from the dependent objects: The Observer pattern enables you to notify dependent objects of changes in the state of the subject object without tightly coupling the objects. This can help improve the maintainability and scalability of your code.
2. When you have multiple objects that need to respond to changes in the state of the subject object: The Observer pattern provides a way to automatically update all dependent objects when a change occurs in the subject object. This can help reduce code redundancy and improve the efficiency of your code.

3. When you have a user interface that needs to respond to changes in the state of the underlying data model: The Observer pattern is commonly used in user interface development to update the UI components when the underlying data model changes. This can help improve the user experience and make the UI more responsive.
4. When you need to implement event-driven architectures: The Observer pattern is often used in event-driven architectures, where events are generated by one or more objects and need to be handled by other objects in the system. The Observer pattern provides a way to notify the interested parties when an event occurs.

When it should not be used:

While the Observer pattern can be useful in many situations, there are also scenarios where it may not be the best choice. Here are some scenarios where the Observer pattern may not be suitable:

1. When the number of observers is fixed: If the number of observers is fixed and known in advance, it may be more appropriate to use the Mediator pattern. The Mediator pattern enables communication between objects, but unlike the Observer pattern, it does not allow for dynamic registration and removal of observers.
2. When the observers need to know the order in which events occurred: The Observer pattern does not provide any guarantees on the order in which events are processed by the observers. If the observers need to know the order in which events occurred, it may be more appropriate to use the Chain of Responsibility pattern.
3. When the observer objects are tightly coupled with the subject object: If the observer objects are tightly coupled with the subject object, it may be better to use a simpler, more direct approach for updating the observer objects. For example, you could use method calls to update the observer objects instead of relying on an Observer pattern.
4. When the observer objects need to be updated with only specific changes: If the observer objects only need to be updated with specific changes in the subject object's state, it may be more appropriate to use the Command pattern. The Command pattern provides a way to encapsulate changes to an object's state and apply them selectively to observer objects.

7. **Pattern Name:** Singleton

Common Uses :
The Singleton pattern is commonly used in situations where you need to ensure that only one instance of a class is created and that it can be accessed globally. For example, a database connection manager or a logging service could be implemented using the Singleton pattern.

Energy Saving Rationale :
The Singleton pattern can be beneficial for energy efficiency in situations where creating and initialising multiple instances of a class would be resource-intensive. By ensuring that only one instance of the class is created and shared among multiple components, the overall resource consumption can be reduced.

When it should be used :
The Singleton pattern should be used when you need to ensure that only one instance of a class is created and that it can be accessed globally. It is useful in situations where resource consumption is a concern and where sharing a single instance of the class can lead to more efficient resource utilisation.

When it should not be used :
The Singleton pattern may not be suitable in situations where multiple instances of the class are required, or where there is a risk of introducing performance bottlenecks due to contention for the shared instance. Additionally, in situations where the singleton instance needs to be frequently created and destroyed, the overhead of maintaining the singleton instance may outweigh the benefits.

8. **Pattern Name:** Builder

Common Uses :
The Builder pattern is commonly used in situations where you need to create objects with multiple configuration options that can be set in any order. For example, a GUI component that allows users to customise the appearance of a chart could be implemented using the Builder pattern.

Energy Saving Rationale :
The Builder pattern can be beneficial for energy efficiency in situations where a large number of configuration options are available, and it is not necessary to set all of them for every object that is created. By allowing the configuration options to be set in any order, the Builder pattern can reduce the number of unnecessary computations required to create objects.

When it should be used :
The Builder pattern should be used when you need to create complex objects with many configuration options that can be set in any order. It is useful in situations where there is a large number of possible configuration options and where the same configuration options may be reused to create multiple objects.

When it should not be used :
The Builder pattern may not be suitable in situations where the configuration options for the object are limited, or where there is a limited number of objects that need to be created. Additionally, in situations where the configuration options are frequently changing, the overhead of maintaining the builder objects may outweigh the benefits.

9. **Pattern Name:** Proxy Pattern

The proxy pattern in software design and analysis allows the creation of an intermediary object that acts as a proxy/substitute or placeholder of the original object.

Common uses:

It is commonly used to protect the original object by using its proxy which provides the same functionality.

Remote proxy: in the network, a proxy will be an intermediary server. Traffic flows through it, from server to client, request and response. The Proxy server provides caching firewall and filtering.

Protection proxy: IT controls access to the real object, proxy acts as a gatekeeper, allowing only authorized clients to the real object.

Cache proxy: store the results in a proxy which are frequently searched or accessed and the proxy can provide those results instead of requesting a real server/object every time.

Logging proxy: can be used to log information flow(request and response) between client and server.

Energy saving rationale:

This pattern avoids duplication of the creation of objects leading to less memory usage and optimizing the performance of the application. Hence, less energy consumption by the app. It can be used in situations to save energy consumption. For example, caching proxy stores results of resource-consuming requests, reducing the number of expensive operations and saving energy. It also reduces network traffic.

When it should be used:

When we want to increase the functionalities of an object without modifying it, we can use a proxy addition to it to provide additional functionalities. Moreover, to provide security, privacy or protection, proxies are commonly used. It is also used for providing caching. Moreover, when it is too expensive to access the real object, you can access a proxy unless the real object is really needed.

When it should not be used:

Proxy patterns can give rise to bottlenecks so they should not be used where there is the possibility of a bottleneck. As requests are going through a proxy server, it can be a bottleneck stopping or delaying the requests to go to the original server. Adding proxies can also lead to over-abstraction (too many proxies, too many layers of abstraction) creating more complexity in code.

10. Pattern Name: Flyweight Pattern

If you have to create a lot of objects, like books in a bookstore, try avoiding the properties/states inside the objects which will be rarely used, so lightweight objects can be created. You move those attributes to a new class called flyweight class which is immutable. So it splits the initial state of the object to two, intrinsic mutable and extrinsic immutable.

Common uses:

It is usually used in GUI applications, and game development for creation and rendering of large numbers of objects.

Energy saving rationale:

Using flyweight patterns improves performance and reduces resource consumption, and leads to less energy consumption.

When it should be used:

It is used to minimize memory(RAM) usage, and improve performance. It is mostly used where a large number of object creation is required and cost to create objects is high in terms of time and memory.

When it should not be used:

Flyweight is of no use when there are fewer objects created, it should only be used when a program must support a huge number of similar objects. Also, when objects have unique properties, flyweight patterns can not be used.

11. **Pattern Name:** Strategy

Common uses:
In object oriented programming, the Strategy pattern is a behavioural design pattern that allows the developer to define a family of algorithms, encapsulating each one as an object, and making them interchangeable. Strategy pattern lets the algorithm vary independently from the clients that use it. Thus strategy pattern is usually used in situations where multiple algorithms or strategies can be used to perform a task, and the selection of the appropriate algorithm or strategy depends on the runtime context. This makes the code flexible, accommodating and reusable.
For example, take the case of an e-commerce website that contains payment processing through different payment gateways like Paypal, credit card, debit card etc,. Here, a strategy pattern can be used such that a PaymentStrategy interface is created and it has different concrete strategies for each payment gateway processing logic in separate classes.

Energy saving rationale:
Though the strategy pattern doesn't directly lead to energy saving, it can help provide a way to dynamically switch between a family of algorithms at runtime that may have different energy consumption characteristics. This could be leveraged to make the system energy efficient by picking the algorithm with less overhead, thereby optimising the energy consumption of the software. For example, in sorting algorithms, the quicksort algorithm is efficient if the size of the input is very large. But, insertion sort is more efficient than quick sort in case of small arrays because quicksort has extra overhead from the recursive function calls. Therefore, strategy pattern could be employed to pick a sorting algorithm based on the n value (number of items to be sorted). Strategy pattern also leads to less code rewrite and could aid in maintainability.

When it should be used:
The Strategy pattern is used in situations where we want to provide clients with flexibility in selecting the appropriate algorithm at runtime while keeping the algorithm details encapsulated and modular while also helping in code reusability.
This pattern is useful when:

- We want to provide clients with the flexibility to switch between different algorithms without modifying client code.
- There are a lot of similar classes that only differ in the way they execute some behaviour.
- There are multiple algorithms that can perform a similar task, but the choice of algorithm depends on the runtime context.
- Separation of concerns is of prime importance in our application.

When it should not be used:
The strategy pattern is not useful when we don't have multiple algorithms performing the same task. It is also not beneficial in cases when the algorithm to be used by the client is

decided at compile time itself. Additionally, strategy pattern is not useful if the overhead of creating strategy objects and maintaining them outweighs the benefits offered by the pattern.

12. Pattern Name: Template method

Common uses:
In object oriented programming, template method pattern is a behavioural design pattern that defines the skeleton of an algorithm, deferring some steps to subclasses. Template method then lets subclasses redefine certain steps of the algorithm without changes to the algorithm structure.
Thus in simple words, if we want to customise only particular steps of an algorithm but not the whole algorithm or its structure, a template method pattern is the right choice. Template method pattern is most commonly used in frameworks, where each implements the invariant parts of a domain while giving clients the ability to redefine for customisation.

Energy saving rationale:
Though the template method pattern doesn't directly result in energy saving, for complex projects it helps in keeping the application code reusable and free of code duplication, thereby making sure developers write meaningful code only. Thus it might indirectly result in less energy, resource consumption and better maintainability of the application.

When it should be used:
The template method pattern is useful in cases when:

- We want the control to change things be restricted in subclasses, thereby limiting only permitted steps of the workflow to be changed.
- We want to make it easy to customise an algorithm without having to modify the existing code (well defined framework with some customisations).
- Flexibility, composability and reuse are of prime importance in a complex application.

Template method pattern is also very useful in code reusability as we define the workflow of the algorithm only once in the abstract class and reuse it.

When it should not be used:
The template method isn't really useful and would not be a good fit if we don't have algorithms with common structure in our application. Templating could also be counterproductive if we don't have enough complex behaviours in our application. Since template method pattern allows only to customise particular steps of an algorithm, it might not be very helpful in cases where we need the entire behaviour of the algorithm to be modified. Strategy pattern will be helpful in this case.

13. **Pattern Name:** Command

Common uses:
The command pattern is a behavioral design pattern that encapsulates a request as an object, thereby allowing for the parameterization of clients with different requests, and the ability to queue or log requests. It is used to implement procedures, transactions, undo functionality, and macros.

Energy saving rationale:
The command pattern can help optimize energy usage by allowing for the queuing of requests, which can reduce the number of times a resource is accessed, thereby reducing energy consumption. Additionally, the pattern can facilitate the implementation of undo functionality, which can reduce the need for repetitive actions that consume energy.

When it should be used:
The command pattern is commonly used when:

- You want to separate the request and the implementation of that request.
- You want to be able to queue requests or log requests for future use.
- You want to provide undo functionality.
- You want to implement a transactional system.

When it should not be used:
The command pattern may not be necessary for simple systems where there are only a few requests to be implemented. It may also not be necessary if a system does not require undo functionality or the ability to queue or log requests. Finally, if the implementation of the requests is straightforward and does not require separation, the command pattern may not be necessary.

14. **Pattern Name:** Composite

Common uses:
The composite pattern is a structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. It represents a hierarchical tree structure of objects and allows clients to treat individual objects and compositions of objects uniformly.

Energy saving rationale:
The composite pattern can help optimize energy usage by reducing the amount of code required to handle similar objects in a hierarchy. By treating individual objects and composition of objects uniformly, composite enables code reusability, which reduces energy consumption by eliminating the need to write new code for similar objects. Additionally, because the composite pattern allows multiple objects to be treated as a single object, it can reduce the number of resource requests, which can also reduce energy consumption.

When it should be used:
The composite pattern is commonly used when:

- You have a hierarchical structure of objects that can be represented as a tree.
- You want to treat individual objects and compositions of objects uniformly.
- You want to be able to ignore the difference between compositions of objects and individual objects.
- You want to be able to add and remove objects from the tree structure dynamically.

When it should not be used:
The composite pattern may not be necessary for systems where there is no need for hierarchical representation or where individual objects cannot be treated uniformly with compositions of objects. Additionally, if the number of objects in the hierarchy is small and does not need to be modified at runtime, the composite pattern may be overly complex and not necessary. Finally, if the hierarchy of objects is too complex, and the addition or removal of objects at runtime is not required, the composite pattern may not be the best approach.

15. **Pattern Name:** Abstract Factory

Using the interfaces provided by the Abstract Factory Pattern, families of linked or dependent items can be created without defining their concrete classes. The Abstract Factory Pattern enables a client to manufacture a group of related products through an abstract interface without knowing the concrete products that are produced. The client is decoupled from any specifics of the concrete goods in this way.

Common uses:
Abstract Factory pattern is commonly used in object-oriented software engineering to provide an interface for creating related or dependent objects, without specifying their concrete classes.

Energy saving rationale:
The Abstract Factory provides a method for building groups of connected or interdependent objects without defining their concrete classes. This pattern saves energy by promoting modularity and reducing code duplication, which leads to a more effective and maintainable software system. The pattern can prevent duplicating object creation code throughout the system by developing a single abstract factory class that encompasses the production of a family of objects. This makes the system easier to manage and modify since the object creation process can be altered in a single place and have those modifications take effect throughout the system.

When it should be used:
The Abstract Factory pattern is should be used when:

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

When it should not be used:
Extending Abstract Factories to produce new kinds of products is not easy. That happens because the Abstract Factory's interface has fixed the set of products that can be created. Therefore, supporting new kinds of products requires extending the factory interface, which involves changing the Abstract Factory class and all of its subclasses. So, this pattern is unsuitable to be used for systems without consistency or no need to enforce consistency among products.
Additionally, the Abstract Factory pattern is designed to create families of related objects, so if there is only one concrete product family, there is no need to use this pattern.

16. **Pattern Name:** Prototype

The Prototype design pattern is commonly used in situations where creating a new object using traditional instantiation is complex and time-consuming, or when creating a large number of identical objects would consume excessive resources.

Common uses:
The Prototype Pattern allows developers to make new instances by copying existing instances. A key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated.

Energy saving rationale:
The Prototype design pattern can reduce the need to create new objects from scratch, which can be resource-intensive, and thus help reduce energy usage in software systems. Instead, it clones or copies existing objects, which can be more effective in terms of memory use and processing time. So, the Prototype design pattern may optimise energy consumption in systems that require frequent object creation and manipulation.
However, it should be noted that sometimes creating objects from scratch may use lower resources. Therefore, skilled developers with deep knowledge of resource usage are needed when considering using this pattern.

When it should be used:
The prototype design pattern is beneficial to be used when:

- The Prototype pattern is useful in situations where the class hierarchy of objects is too complex. So, creating an object through cloning allows for a more flexible and efficient approach
- When object type is unknown, cloning will be the option for the client to generate objects.

When it should not be used:
The prototype design pattern is not recommended to be used when:

- Prototype should not be used when a system must create new objects of many types in a complex class hierarchy.
- Prototype should not be used when making a copy of an object is too complicated. For example when the cost of cloning is high, or when deep copy is required.
- Prototype design also not recommended to be used for unique or simple objects. As this approach will only add an unnecessary complexity and high number of objects.