Coding

Contents

[Programming Paradigms 5](#_Toc499100562)

[1. Object Oriented 5](#_Toc499100563)

[2. Functional 5](#_Toc499100564)

[3. Declarative 5](#_Toc499100565)

[4. Aspect-oriented 5](#_Toc499100566)

[Deployment Strategies 5](#_Toc499100567)

[5. Recreate 5](#_Toc499100568)

[6. Ramped (also known as rolling-update or incremental) 6](#_Toc499100569)

[7. Blue/Green 6](#_Toc499100570)

[8. Canary 6](#_Toc499100571)

[9. A/B testing 6](#_Toc499100572)

[10. Shadow 6](#_Toc499100573)

[Development Methodologies 6](#_Toc499100574)

[11. DevOps 6](#_Toc499100575)

[12. Agile 6](#_Toc499100576)

[13. Extreme 6](#_Toc499100577)

[14. Lean 6](#_Toc499100578)

[Eliminate waste 7](#_Toc499100579)

[Amplify learning 7](#_Toc499100580)

[Decide as late as possible 7](#_Toc499100581)

[Deliver as fast as possible 7](#_Toc499100582)

[Empower the team 8](#_Toc499100583)

[Build integrity in 8](#_Toc499100584)

[See the whole 8](#_Toc499100585)

[15. Kaizen 9](#_Toc499100586)

[16. Test-driven Development 9](#_Toc499100587)

[17. Continuous Test-driven Development 9](#_Toc499100588)

[18. Behaviour-driven development 9](#_Toc499100589)

[19. Waterfall 9](#_Toc499100590)

[20. Cowboy 9](#_Toc499100591)

[21. Abstraction Principle 9](#_Toc499100592)

[22. Big Design Up Front 9](#_Toc499100593)

[23. Continuous Integration 9](#_Toc499100594)

[24. Model Driven Architecture 10](#_Toc499100595)

[25. Open Source 10](#_Toc499100596)

[26. Separation of Concern. Encapsulation, Modular 10](#_Toc499100597)

[Separation of content and presentation 10](#_Toc499100598)

[Development Concepts/Techniques 10](#_Toc499100599)

[27. Code Generation 10](#_Toc499100600)

[28. Pairing 10](#_Toc499100601)

[29. Code Review 10](#_Toc499100602)

[30. Static Code Analysis 11](#_Toc499100603)

[31. Source Code Repo 11](#_Toc499100604)

[32. Freedom and Responsibility 11](#_Toc499100605)

[33. DevOpsDays 11](#_Toc499100606)

[34. State vs Migrations-based deployments 11](#_Toc499100607)

[Architectural Patterns 12](#_Toc499100608)

[35. Blackboard System 12](#_Toc499100609)

[36. Broker Pattern 12](#_Toc499100610)

[37. Event-driven architecture 12](#_Toc499100611)

[38. Implicit invocation 12](#_Toc499100612)

[39. Microservices 12](#_Toc499100613)

[40. Layers 12](#_Toc499100614)

[41. Model-View-Whatever 12](#_Toc499100615)

[42. Entity Component System 12](#_Toc499100616)

[43. Multitier Architecture 12](#_Toc499100617)

[44. Naked Objects 12](#_Toc499100618)

[45. Operational Data Store 12](#_Toc499100619)

[46. Peer-to-Peer 12](#_Toc499100620)

[47. Pipe and Filter Architecture 12](#_Toc499100621)

[48. Service Oriented Architecture 12](#_Toc499100622)

[49. Space Based Architecture 12](#_Toc499100623)

[50. Model-Driven Architecture 12](#_Toc499100624)

[Solid Principles 12](#_Toc499100625)

[51. Single Responsibility 12](#_Toc499100626)

[52. Open Closed 12](#_Toc499100627)

[53. Liskov Substitution 12](#_Toc499100628)

[54. Interface Segregation 12](#_Toc499100629)

[55. Dependency Inversion or Inversion of Control 12](#_Toc499100630)

[Other Principles/Techniques 13](#_Toc499100631)

[56. DRY/ Principle of Abstraction 13](#_Toc499100632)

[57. "Program to an 'interface', not an 'implementation'." 13](#_Toc499100633)

[58. Composition over inheritance: "Favour 'object composition' over 'class inheritance'." 13](#_Toc499100634)

[59. Generics / Parameterised Types 13](#_Toc499100635)

[60. Pseudo code 13](#_Toc499100636)

[61. Object Modelling 13](#_Toc499100637)

[62. Extensibility 13](#_Toc499100638)

[Creational Design Patterns 13](#_Toc499100639)

[63. Abstract factory (GoF) 13](#_Toc499100640)

[64. Builder (GoF) 13](#_Toc499100641)

[65. Dependency Injection (supports Dependency Inversion) 13](#_Toc499100642)

[66. Factory Method (GoF) 14](#_Toc499100643)

[67. Lazy initialization 14](#_Toc499100644)

[68. Multiton 14](#_Toc499100645)

[69. Object Pool 14](#_Toc499100646)

[70. Prototype (GoF) 14](#_Toc499100647)

[71. Singleton (GoF) 14](#_Toc499100648)

[Structural Design Patterns 14](#_Toc499100649)

[72. Adapter, Wrapper, or Translator (GoF) 14](#_Toc499100650)

[73. Bridge/Handle/Body (GoF) 14](#_Toc499100651)

[74. ? Decorator (GoF) 15](#_Toc499100652)

[75. Extension Object 16](#_Toc499100653)

[76. ? Façade (GoF) 16](#_Toc499100654)

[77. ? Flyweight (GoF) 16](#_Toc499100655)

[78. ? Front controller 16](#_Toc499100656)

[79. ? Marker 16](#_Toc499100657)

[80. Module 16](#_Toc499100658)

[81. ? Proxy 16](#_Toc499100659)

[82. Twin 16](#_Toc499100660)

[83. Composite (GoF) 16](#_Toc499100661)

[Behavioural patterns 16](#_Toc499100662)

[84. ? Blackboard 16](#_Toc499100663)

[85. Chain of Responsibility 16](#_Toc499100664)

[86. Command 17](#_Toc499100665)

[87. ? Interpreter 17](#_Toc499100666)

[88. Iterator 17](#_Toc499100667)

[89. ? Mediator 17](#_Toc499100668)

[90. ? Memento 17](#_Toc499100669)

[91. ? Null Object 17](#_Toc499100670)

[92. Observer/Publish Subscribe 17](#_Toc499100671)

[93. ? Servant 17](#_Toc499100672)

[94. ? Specification 17](#_Toc499100673)

[95. ? State 17](#_Toc499100674)

[96. Strategy 17](#_Toc499100675)

[97. Template Method 18](#_Toc499100676)

[98. ? Visitor 18](#_Toc499100677)

[99. Repository Pattern 18](#_Toc499100678)

[100. Plugin/Microkernel 18](#_Toc499100679)

[Concurrency patterns 18](#_Toc499100680)

[101. ? Active Object 18](#_Toc499100681)

[102. ? Balking 18](#_Toc499100682)

[103. Binding properties 18](#_Toc499100683)

[104. ? Blockchain 18](#_Toc499100684)

[105. ? Double-checked locking 19](#_Toc499100685)

[106. ? Event-based asynchronous 19](#_Toc499100686)

[107. ? Guarded Suspension 19](#_Toc499100687)

[108. ? Join 19](#_Toc499100688)

[109. Lock 19](#_Toc499100689)

[110. ? Messaging design pattern (MDP) 19](#_Toc499100690)

[111. ? Monitor object 19](#_Toc499100691)

[112. ? Reactor 19](#_Toc499100692)

[113. ? Read-write lock 19](#_Toc499100693)

[114. ? Scheduler 19](#_Toc499100694)

[115. Thread Pool 19](#_Toc499100695)

[116. Thread-Specific Storage 19](#_Toc499100696)

[Anti-Patterns 20](#_Toc499100697)

[117. Service Locator Pattern 20](#_Toc499100698)

[118. Bikeshedding 20](#_Toc499100699)

[Testing 20](#_Toc499100700)

[119. Unit Testing 20](#_Toc499100701)

[120. Faking 20](#_Toc499100702)

[121. Mocking 20](#_Toc499100703)

[122. Stubbing 20](#_Toc499100704)

[Formatting 21](#_Toc499100705)

[123. Cases 21](#_Toc499100706)

[124. Use smart variables 21](#_Toc499100707)

[125. Avoid magic numbers 21](#_Toc499100708)

[126. Methods and classes for everything 22](#_Toc499100709)

[127. Comment and comment well 22](#_Toc499100710)

[128. Don’t reinvent the wheel 23](#_Toc499100711)

[Uncategorised 24](#_Toc499100712)

[129. Law of Demetor 24](#_Toc499100713)

[130. High Cohesion 24](#_Toc499100714)

[131. Loose Coupling 24](#_Toc499100715)

[132. Modular 24](#_Toc499100716)

[133. Fluent Interface 24](#_Toc499100717)

[134. Encapsulation 24](#_Toc499100718)

[135. Session 25](#_Toc499100719)

[136. Caching 25](#_Toc499100720)

[137. Logging 25](#_Toc499100721)

[138. Setting management 25](#_Toc499100722)

[139. Timing 25](#_Toc499100723)

[140. Object to Object Mapping (Auto Mapping) 25](#_Toc499100724)

[141. Domain Layer 25](#_Toc499100725)

[Entities 25](#_Toc499100726)

[Value Objects 25](#_Toc499100727)

[Repositories 25](#_Toc499100728)

[Domain Services 25](#_Toc499100729)

[Specifications 25](#_Toc499100730)

[Unit of Work 25](#_Toc499100731)

[Domain Events (EventBus) 25](#_Toc499100732)

[Data Filters 25](#_Toc499100733)

[142. Application Layer 25](#_Toc499100734)

[Application Services 25](#_Toc499100735)

[Data Transfer Objects 25](#_Toc499100736)

[Validating Data Transfer Objects 25](#_Toc499100737)

[Authorization 25](#_Toc499100738)

[Feature Management 25](#_Toc499100739)

[Audit Logging 25](#_Toc499100740)

[143. Distributed Service Layer 25](#_Toc499100741)

[ASP.NET Web API 25](#_Toc499100742)

[Web API Controllers 25](#_Toc499100743)

[Dynamic Web API Layer 25](#_Toc499100744)

[OData Integration 25](#_Toc499100745)

[Swagger UI Integration 25](#_Toc499100746)

[144. Presentation Layer 25](#_Toc499100747)

[ASP.NET MVC 25](#_Toc499100748)

[MVC Controllers 25](#_Toc499100749)

[MVC Views 25](#_Toc499100750)

[Handling Exceptions 25](#_Toc499100751)

[Localization 25](#_Toc499100752)

[Navigation 25](#_Toc499100753)

[Embedded Resources 25](#_Toc499100754)

[Javascript API 25](#_Toc499100755)

[CSRF/XSRF Protection 25](#_Toc499100756)

[145. Infrastructure Layer 26](#_Toc499100757)

[Background Services 26](#_Toc499100758)

[Background Jobs and Workers 26](#_Toc499100759)

[Hangfire Integration 26](#_Toc499100760)

[Quartz Integration 26](#_Toc499100761)

[Real Time Services 26](#_Toc499100762)

[Notification System 26](#_Toc499100763)

[SignalR Integration 26](#_Toc499100764)

[Object-Relational Mapping 26](#_Toc499100765)

[EntityFramework Integration 26](#_Toc499100766)

[EntityFramework Core Integration 26](#_Toc499100767)

[NHibernate Integration 26](#_Toc499100768)

[Dapper Integration NEW 26](#_Toc499100769)

[Left Out 26](#_Toc499100770)

[146. Resource acquisition is initialization (RAII) 26](#_Toc499100771)

[References 26](#_Toc499100772)

[https://en.wikipedia.org/wiki/Abstract\_factory\_pattern 26](#_Toc499100773)

[http://www.oodesign.com/ 26](#_Toc499100774)

[https://en.wikipedia.org/wiki/Software\_design\_pattern 26](#_Toc499100775)

[https://en.wikipedia.org/wiki/Architectural\_pattern 26](#_Toc499100776)

[Books 26](#_Toc499100777)

[https://en.wikipedia.org/wiki/The\_Pragmatic\_Programmer 26](#_Toc499100778)

[https://en.wikipedia.org/wiki/Design\_Patterns by Gang of Four 26](#_Toc499100779)

[https://en.wikipedia.org/wiki/Code\_Complete 26](#_Toc499100780)

[https://martinfowler.com/books/eaa.html by Martin Fowler 26](#_Toc499100781)

# Programming Paradigms

## Object Oriented

## Functional

## Declarative

## Aspect-oriented

# Deployment Strategies

## Recreate

Version A is terminated then version B is rolled out.

The recreate strategy is a dummy deployment which consists of shutting down version A then deploying version B after version A is turned off. This technique implies downtime of the service that depends on both shutdown and boot duration of the application.

Pros:

* Easy to setup.
* Application state entirely renewed.

Cons:

* High impact on the user, expect downtime that depends on both shutdown and boot duration of the application.

## Ramped (also known as rolling-update or incremental)

Version B is slowly rolled out and replacing version A.

The ramped deployment strategy consists of slowly rolling out a version of an application by replacing instances one after the other until all the instances are rolled out. It usually follows the following process: with a pool of version A behind a load balancer, one instance of version B is deployed. When the service is ready to accept traffic, the instance is added to the pool. Then, one instance of version A is removed from the pool and shut down.

Depending on the system taking care of the ramped deployment, you can tweak the following parameters to increase the deployment time:

* Parallelism, max batch size: Number of concurrent instances to roll out.
* Max surge: How many instances to add in addition of the current amount.
* Max unavailable: Number of unavailable instances during the rolling update procedure.

Pros:

* Easy to set up.
* Version is slowly released across instances.
* Convenient for stateful applications that can handle rebalancing of the data.

Cons:

* Rollout/rollback can take time.
* Supporting multiple APIs is hard.
* No control over traffic.

## Blue/Green

Version B is released alongside version A, then the traffic is switched to version B.

The blue/green deployment strategy differs from a ramped deployment, version B (green) is deployed alongside version A (blue) with exactly the same amount of instances. After testing that the new version meets all the requirements the traffic is switched from version A to version B at the load balancer level.

Pros:

* Instant rollout/rollback.
* Avoid versioning issue, the entire application state is changed in one go.

Cons:

* Expensive as it requires double the resources.
* Proper test of the entire platform should be done before releasing to production.
* Handling stateful applications can be hard.

## Canary

Version B is released to a subset of users, then proceed to a full rollout.

A canary deployment consists of gradually shifting production traffic from version A to version B. Usually the traffic is split based on weight. For example, 90 percent of the requests go to version A, 10 percent go to version B.

This technique is mostly used when the tests are lacking or not reliable or if there is little confidence about the stability of the new release on the platform.

Pros:

* Version released for a subset of users.
* Convenient for error rate and performance monitoring.
* Fast rollback.

Con:

* Slow rollout.

## A/B testing

Version B is released to a subset of users under specific condition.

A/B testing deployments consists of routing a subset of users to a new functionality under specific conditions. It is usually a technique for making business decisions based on statistics, rather than a deployment strategy. However, it is related and can be implemented by adding extra functionality to a canary deployment so we will briefly discuss it here.

This technique is widely used to test conversion of a given feature and only roll-out the version that converts the most.

Here is a list of conditions that can be used to distribute traffic amongst the versions:

* By browser cookie
* Query parameters
* Geolocalisation
* Technology support: browser version, screen size, operating system, etc.
* Language

Pros:

* Several versions run in parallel.
* Full control over the traffic distribution.

Cons:

* Requires intelligent load balancer.
* Hard to troubleshoot errors for a given session, distributed tracing becomes mandatory.

## Shadow

Version B receives real-world traffic alongside version A and doesn’t impact the response.

A shadow deployment consists of releasing version B alongside version A, fork version A’s incoming requests and send them to version B as well without impacting production traffic. This is particularly useful to test production load on a new feature. A rollout of the application is triggered when stability and performance meet the requirements.

This technique is fairly complex to setup and needs special requirements, especially with egress traffic. For example, given a shopping cart platform, if you want to shadow test the payment service you can end-up having customers paying twice for their order. In this case, you can solve it by creating a mocking service that replicates the response from the provider.

Pros:

* Performance testing of the application with production traffic.
* No impact on the user.
* No rollout until the stability and performance of the application meet the requirements.

Cons:

* Expensive as it requires double the resources.
* Not a true user test and can be misleading.
* Complex to setup.
* Requires mocking service for certain cases.

# Development Methodologies

## DevOps

Calms

1. Culture
2. Automation
3. Lean
4. Measurement
5. Sharing

What to do with legacy QA? <https://www.infoworld.com/article/3238085/devops/legacy-qa-in-the-devops-age-reskill-them-or-release-them.html>

## Agile

## Extreme

Advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, code simplicity and clarity, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers.[2][3][4] The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels. As an example, code reviews are considered a beneficial practice; taken to the extreme, code can be reviewed continuously, i.e. the practice of pair programming.

## Lean

Lean development can be summarized by seven principles, very close in concept to lean manufacturing principles:

### Eliminate waste

The following are wastes:

Building the wrong feature or product

Mismanaging the backlog

Rework

Unnecessarily complex solutions

Extraneous cognitive load

Psychological distress

Waiting/multitasking

Knowledge loss

Ineffective communication.

### Amplify learning

Software development is a continuous learning process based on iterations when writing code. Software design is a problem solving process involving the developers writing the code and what they have learned. Software value is measured in fitness for use and not in conformance to requirements.

Instead of adding more documentation or detailed planning, different ideas could be tried by writing code and building. The process of user requirements gathering could be simplified by presenting screens to the end-users and getting their input. The accumulation of defects should be prevented by running tests as soon as the code is written.

The learning process is sped up by usage of short iteration cycles – each one coupled with refactoring and integration testing. Increasing feedback via short feedback sessions with customers helps when determining the current phase of development and adjusting efforts for future improvements. During those short sessions both customer representatives and the development team learn more about the domain problem and figure out possible solutions for further development. Thus the customers better understand their needs, based on the existing result of development efforts, and the developers learn how to better satisfy those needs. Another idea in the communication and learning process with a customer is set-based development – this concentrates on communicating the constraints of the future solution and not the possible solutions, thus promoting the birth of the solution via dialogue with the customer.

### Decide as late as possible

Delaying decisions as much as possible until they can be made based on facts and not on uncertain assumptions and predictions.

### Deliver as fast as possible

In the era of rapid technology evolution, it is not the biggest that survives, but the fastest. The sooner the end product is delivered without major defects, the sooner feedback can be received, and incorporated into the next iteration. The shorter the iterations, the better the learning and communication within the team. With speed, decisions can be delayed. Speed assures the fulfilling of the customer's present needs and not what they required yesterday. This gives them the opportunity to delay making up their minds about what they really require until they gain better knowledge. Customers value rapid delivery of a quality product.

### Empower the team

There has been a traditional belief in most businesses about the decision-making in the organization – the managers tell the workers how to do their own job. In a "Work-Out technique", the roles are turned – the managers are taught how to listen to the developers, so they can explain better what actions might be taken, as well as provide suggestions for improvements. The lean approach follows the Agile Principle[7] "find good people and let them do their own job,"[8] encouraging progress, catching errors, and removing impediments, but not micro-managing.

Another mistaken belief has been the consideration of people as resources. People might be resources from the point of view of a statistical data sheet, but in software development, as well as any organizational business, people do need something more than just the list of tasks and the assurance that they will not be disturbed during the completion of the tasks. People need motivation and a higher purpose to work for – purpose within the reachable reality, with the assurance that the team might choose its own commitments. The developers should be given access to the customer; the team leader should provide support and help in difficult situations, as well as ensure that scepticism does not ruin the team’s spirit.

### Build integrity in

The customer needs to have an overall experience of the System. This is the so-called perceived integrity: how it is being advertised, delivered, deployed, accessed, how intuitive its use is, its price and how well it solves problems.

Conceptual integrity means that the system’s separate components work well together as a whole with balance between flexibility, maintainability, efficiency, and responsiveness. This could be achieved by understanding the problem domain and solving it at the same time, not sequentially. The needed information is received in small batch pieces – not in one vast chunk - preferably by face-to-face communication and not any written documentation. The information flow should be constant in both directions – from customer to developers and back, thus avoiding the large stressful amount of information after long development in isolation.

### See the whole

Software systems nowadays are not simply the sum of their parts, but also the product of their interactions. Defects in software tend to accumulate during the development process – by decomposing the big tasks into smaller tasks, and by standardizing different stages of development, the root causes of defects should be found and eliminated. The larger the system, the more organizations that are involved in its development and the more parts are developed by different teams, the greater the importance of having well defined relationships between different vendors, in order to produce a system with smoothly interacting components. During a longer period of development, a stronger subcontractor network is far more beneficial than short-term profit optimizing, which does not enable win-win relationships.

Lean thinking has to be understood well by all members of a project, before implementing in a concrete, real-life situation. "Think big, act small, fail fast; learn rapidly"[9] – these slogans summarize the importance of understanding the field and the suitability of implementing lean principles along the whole software development process. Only when all of the lean principles are implemented together, combined with strong "common sense" with respect to the working environment, is there a basis for success in software development.

## Kaizen

Japanese. Kaizen refers to activities that continuously improve all functions and involve all employees

## Test-driven Development

Relies on the repetition of a very short development cycle: Requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements.

## Continuous Test-driven Development

Automated Test-driven Development. Feature available on GitHub and visual studio team services.

## Behaviour-driven development

## Waterfall

Old school big design up front.

## Cowboy

Where programmers have autonomy over the development process. This includes control of the project's schedule, languages, algorithms, tools, frameworks and coding style.

A cowboy coder can be a lone developer or part of a group of developers working with minimal process or discipline.

Anti-pattern

## Abstraction Principle

Basic dictum that aims to reduce duplication of information in a program.

## Big Design Up Front

The program's design is to be completed and perfected before that program's implementation is started. It is often associated with the waterfall model of software development.

## Continuous Integration

Security Analysis. (<https://blog.aquasec.com/container-security-automation-keeping-up-with-the-devops-crowd> )

## Model Driven Architecture

The model-driven architecture approach defines system functionality using a platform-independent model (PIM) using an appropriate domain-specific language (DSL).

Then, given a platform model corresponding to CORBA, .NET, the Web, etc., the PIM is translated to one or more platform-specific models (PSMs) that computers can run. This requires mappings and transformations and should be modelled too.

The PSM may use different DSLs or a general purpose language. Automated tools generally perform this translation.

OMG focuses Model-driven architecture on forward engineering, i.e. producing code from abstract, human-elaborated modelling diagrams.

Usually involves Code Generation.

## Open Source

## Separation of Concern. Encapsulation, Modular

### Separation of content and presentation

HTML & CSS

# Development Concepts/Techniques

## Code Generation

## Pairing

Pair Programming.

## Code Review

Simple Definition

A code review is a process where two or more developers visually inspect a set of program code, typically, several times. The code can be a method, a class, or an entire program. The main code-review objectives are:

Best Practice ~ A more efficient, less error-prone, or more elegant way to accomplish a given task.

Error Detection ~ Discovering logical or transitional errors.

Vulnerability Exposure ~ Identifying and averting common vulnerabilities like Cross-Site Scripting [XSS], Injection, Buffer Overflow, Excessive Disclosure, etc. Although many controls are inapplicable and can be ignored, a STIG [e.g., Application Security STIG 4.3] provides an excellent vulnerability checklist.

Malware Discovery ~ This often-overlooked, and very special code-review objective looks for segments of code that appear extraneous, questionable, or flat-out weird. The intent is to discover back doors, Trojans, and time bombs. I say often-overlooked because the very idea of malware and malicious intent may ring overly dramatic to some developers. However - particularly in today’s peril-ridden world - malevolent code is a very real threat and should not be overlooked… especially by USG agencies and departments like the DoD.

Of the four objectives, malware is the only one that requires human detection. A program containing an obvious back door can be scanned using a tool like Fortify and come out looking as clean as the driven snow.

Tools: Atlassian Crucible

## Static Code Analysis

Options:

StyleCop,

NDepend in continuous integration

## Source Code Repo

Options: Bitbucket, VSTS, GitKraken

## Freedom and Responsibility

The Netflix culture of freedom and responsibility empowers engineers to craft solutions using whatever tools they feel are best suited to the task.

## DevOpsDays

DevOpsDays for internal teams, featuring demos, open labs, lightning talks, breakout sessions, and guest keynotes.

## State vs Migrations-based deployments

https://www.red-gate.com/hub/product-learning/sql-source-control/moving-from-application-automation-to-true-devops-by-including-the-database

# Architectural Patterns

## Blackboard System

## Broker Pattern

## Event-driven architecture

## Implicit invocation

## Microservices

## Layers

## Model-View-Whatever

## Entity Component System

## Multitier Architecture

## Naked Objects

## Operational Data Store

## Peer-to-Peer

## Pipe and Filter Architecture

## Service Oriented Architecture

## Space Based Architecture

## Model-Driven Architecture

Provides a set of guidelines for the structuring of specifications, which are expressed as models. Model-driven architecture is a kind of domain engineering, and supports model-driven engineering of software systems

# Solid Principles

## Single Responsibility

A class should have one and only one reason to change, meaning that a class should have only one job.

## Open Closed

Objects or entities should be open for extension, but closed for modification.

## Liskov Substitution

Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

## Interface Segregation

A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.

## Dependency Inversion or Inversion of Control

Entities must depend on abstractions not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

“Hollywood Principle” – “don’t call me, I’ll call you.”

# Other Principles/Techniques

## DRY/ Principle of Abstraction

Do not repeat yourself

Code Reuse, Loose Coupling

## "Program to an 'interface', not an 'implementation'."

## Composition over inheritance: "Favour 'object composition' over 'class inheritance'."

## Generics / Parameterised Types

## Pseudo code

## Object Modelling

## Extensibility

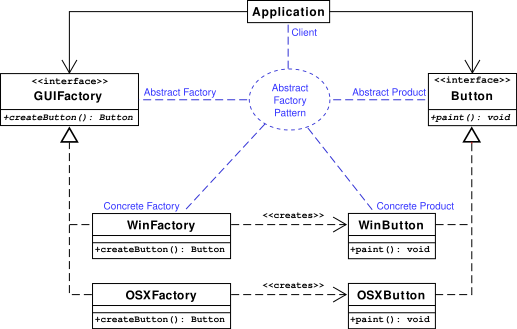
Extensibility is a software engineering and systems design principle where the implementation takes future growth into consideration. The term extensibility can also be seen as a systemic measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. The central theme is to provide for change – typically enhancements – while minimizing impact to existing system functions.

# Creational Design Patterns

## Abstract factory (GoF)

[[1]](#_https://en.wikipedia.org/wiki/Abstr) Provides a way to encapsulate a group of individual [factories](https://en.wikipedia.org/wiki/Factory_object) that have a common theme without specifying their concrete classes.

Used in ADO.net.



## Builder (GoF)

Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.

## Dependency Injection (supports Dependency Inversion)

A class accepts the objects it requires from an injector instead of creating the objects directly. Allows for dependencies to easily be swapped out for testing purposes.

Used in ASP.net Core Services

Used in Angular Components

High usefulness

## Factory Method (GoF)

uses factory methods to deal with the problem of [creating objects](https://en.wikipedia.org/wiki/Object_creation) without having to specify the exact [class](https://en.wikipedia.org/wiki/Class_(computer_programming)) of the object that will be created.

Used in In ADO.NET, IDbCommand.CreateParameter.

## Lazy initialization

Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the [Proxy](https://en.wikipedia.org/wiki/Proxy_pattern) pattern.

Used in ?

## Multiton

The multiton pattern expands on the singleton concept to manage a [map](https://en.wikipedia.org/wiki/Associative_array) of named instances as key-value pairs.

Used in ?

Low usefulness.

## Object Pool

Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.

Used in System.Threading.ThreadPool and ConnectionPool

## Prototype (GoF)

Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.

## Singleton (GoF)

Ensure a class has only one instance, and provide a global point of access to it.

# Structural Design Patterns

## Adapter, Wrapper, or Translator (GoF)

Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.

## Bridge/Handle/Body (GoF)

Decouple an abstraction from its implementation allowing the two to vary independently.

*// Helps in providing truly decoupled architecture*

**public** **interface** IBridge

{

**void** Function1();

**void** Function2();

}

**public** **class** **Bridge1** : IBridge

{

**public** **void** Function1()

{

Console.WriteLine("Bridge1.Function1");

}

**public** **void** Function2()

{

Console.WriteLine("Bridge1.Function2");

}

}

**public** **class** **Bridge2** : IBridge

{

**public** **void** Function1()

{

Console.WriteLine("Bridge2.Function1");

}

**public** **void** Function2()

{

Console.WriteLine("Bridge2.Function2");

}

}

**public** **interface** IAbstractBridge

{

**void** CallMethod1();

**void** CallMethod2();

}

**public** **class** **AbstractBridge** : IAbstractBridge

{

**public** IBridge bridge;

**public** AbstractBridge(IBridge bridge)

{

**this**.bridge = bridge;

}

**public** **void** CallMethod1()

{

**this**.bridge.Function1();

}

public void CallMethod2()

{

**this**.bridge.Function2();

}

}

## ? Decorator (GoF)

Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to sub-classing for extending functionality.

Used in WPF. Grid.Row.

## Extension Object

Adding functionality to a hierarchy without changing the hierarchy.

## ? Façade (GoF)

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.

## ? Flyweight (GoF)

Use sharing to support large numbers of similar objects efficiently.

## ? Front controller

The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.

## ? Marker

Empty interface to associate metadata with a class.

## Module

Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.

## ? Proxy

Provide a surrogate or placeholder for another object to control access to it

## Twin

Twin allows modelling of multiple inheritance in programming languages that do not support this feature.

## Composite (GoF)

Compose objects into tree structures to represent part-whole hierarchies. Composite lets

clients treat individual objects and compositions of objects uniformly.

A *recursive composition* is a hierarchical structure of elements, that builds "increasingly complex elements out of simpler ones" (pp36). Each node in the structure knows of its own children and its parent. If an operation is to be performed on the whole structure, each node calls the operation on its children (recursively).

# Behavioural patterns

## ? Blackboard

Artificial intelligence pattern for combining disparate sources of data

## 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.

Use for logging handlers to log at one of a variety of levels.

## Command

Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations

## ? Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

## Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

## ? 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, and it allows their interaction to vary independently.

## ? Memento

Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.

## ? Null Object

Avoid null references by providing a default object.

## Observer/Publish Subscribe

Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.

## ? Servant

Define common functionality for a group of classes.

## ? Specification

Re-combinable business logic in a Boolean fashion.

## ? State

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

## Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

The Strategy Design Pattern allows an object to have some or all of its behavior defined in terms of another object which follows a particular interface. A particular instance of this interface is provided to the client when it is instantiated or invoked, providing the concrete behavior to be used. The Strategy design pattern is used extensively to achieve the Single Responsibility Principle, the Explicit Dependencies Principle, and the Dependency Inversion Principle, and is a key to Dependency Injection and the use of Inversion of Control Containers.

## 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.

## ? Visitor

Represent an operation to be performed on the elements of an object structure. Visitor lets a new operation be defined without changing the classes of the elements on which it operates

## Repository Pattern

The Repository Pattern has gained quite a bit of popularity since it was first introduced as a part of Domain-Driven Design in 2004. Essentially, it provides an abstraction of data, so that your application can work with a simple abstraction that has an interface approximating that of a collection. Adding, removing, updating, and selecting items from this collection is done through a series of straightforward methods, without the need to deal with database concerns like connections, commands, cursors, or readers. Using this pattern can help achieve loose coupling and can keep domain objects persistence ignorant. Although the pattern is very popular (or perhaps because of this), it is also frequently misunderstood and misused. There are many different ways to implement the Repository pattern.

## Plugin/Microkernel

Links classes during configuration rather than compilation. Inversion of control.

The microkernel architecture pattern (sometimes referred to as the plug-in architecture pattern) is a natural pattern for implementing product-based applications. A product-based application is one that is packaged and made available for download in versions as a typical third-party product. However, many companies also develop and release their internal business applications like software products, complete with versions, release notes, and pluggable features. These are also a natural fit for t

# Concurrency patterns

## ? Active Object

Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.

## ? Balking

Only execute an action on an object when the object is in a particular state.

## Binding properties

Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.

Used in MVVM and Angular

## ? Blockchain

Decentralized way to store data and agree on ways of processing it in a Merkle tree, optionally using digital signature for any individual contributions.

## ? Double-checked locking

Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual locking logic proceed.

## ? Event-based asynchronous

Addresses problems with the asynchronous pattern that occur in multithreaded programs.

## ? Guarded Suspension

Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.

## ? Join

Join-pattern provides a way to write concurrent, parallel and distributed programs by message passing. Compared to the use of threads and locks, this is a high-level programming model.

## Lock

One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.

## ? Messaging design pattern (MDP)

Allows the interchange of information (i.e. messages) between components and applications.

## ? Monitor object

An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.

## ? Reactor

A reactor object provides an asynchronous interface to resources that must be handled synchronously.

## ? Read-write lock

Allows concurrent read access to an object, but requires exclusive access for write operations.

## ? Scheduler

Explicitly control when threads may execute single-threaded code.

## Thread Pool

A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.

## Thread-Specific Storage

Static or "global" memory local to a thread

# Anti-Patterns

## Service Locator Pattern

Used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central registry known as the "service locator", which on request returns the information necessary to perform a certain task.[1] Note that many[weasel words] consider service locator to actually be an anti-pattern.[2]

## Bikeshedding

bikeshed +‎ -ing. The term was coined as a metaphor to illuminate Parkinson’s Law of Triviality. Parkinson observed that a committee whose job is to approve plans for a nuclear power plant may spend the majority of its time on relatively unimportant but easy-to-grasp issues, such as what materials to use for the staff bikeshed, while neglecting the design of the power plant itself, which is far more important but also far more difficult to criticize constructively. It was popularized in the Berkeley Software Distribution community by Poul-Henning Kamp[1] and has spread from there to the software industry at large.

Futile investment of time and energy in discussion of marginal technical issues.

Procrastination.

# Testing

## Unit Testing

## Faking

Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production.

Fake: We acquire or build a very lightweight implementation of the same functionality as provided by a component that the SUT depends on and instruct the SUT to use it instead of the real.

## Mocking

Mocks are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Mock Object that implements the same interface as an object on which the SUT (System Under Test) depends. We can use a Mock Object as an observation point when we need to do Behavior Verification to avoid having an Untested Requirement (see Production Bugs on page X) caused by an inability to observe side-effects of invoking methods on the SUT.

The main thing to remember about mocks versus stubs is that mocks are just like stubs, but you assert against the mock object, whereas you do not assert against a stub.

## Stubbing

Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'

Stub : This implementation is configured to respond to calls from the SUT with the values (or exceptions) that will exercise the Untested Code (see Production Bugs on page X) within the SUT. A key indication for using a Test Stub is having Untested Code caused by the inability to control the indirect inputs of the SUT

# Formatting

## Cases

PascalCase for classes and methods

camelCase for parameters

\_underscore for private variables?

?

## Use smart variables

Using smart variables is very important if you want to make your code even slightly decipherable after a time away.

The obvious tip is to name the variables based on what they do. So, maybe don’t call the user name string MonkeyWrench – call it UserName.

Where possible, try to make your code read in a manner similar to English. This is something that becomes especially apparent when using Booleans (true or false statements).

Of course, you should also be equally logical when choosing names for methods and classes.

## Avoid magic numbers

In some ways, magic numbers are more of a problem than randomly named variables. These are numbers that you assign special meaning to that are completely arbitrary.

For example, I created an ‘overshoot’ animation from scratch so that a view would slide in from the edge of the screen, overshoot its end destination, and then appear to ‘ping’ back into the correct place.

We know that ‘0’ is left and ‘1’ is right. But does everyone else?

To do this, I allowed the image to overshoot its mark by 30 pixels before pinging back. The question you should ask at this point is ‘why 30’?

A more common example of this might be the old ‘Facing’ variable in a basic 2D game. The player can face left or right and in many cases, we will assign one of these directions to ‘0’ and one of these directions to ‘1’. We know that ‘0’ is left and ‘1’ is right. But does everyone else? Will we still know that in one month, or one year?

What should you do instead? Well, you could create constants. For instance:

JAVASELECT ALL

JAVASELECT ALL

private static final int left = 0;

private static final int right = 1;

Now you can say if (Facing = left) and that is hugely more readable.

Likewise, instead of pinging back at ‘30’ we could ping back at overshootAmount or something similar. This also has the added bonus of allowing us to easily tweak how exaggerated our animations are. We could even make this an option available for the user to change.

## Methods and classes for everything

Create methods and classes wherever possible to break up your code. If you then give those methods logical, readable names, then your code will end up short and easy to follow with the option to dig into the nuts and bolts of each step only as necessary: if this, get this number, then draw picture on screen, then save this file…

If you follow this line of logic, larger methods will be broken down into multiple smaller methods. This not only keeps everything nicely organized on the screen allowing you to handle it in digestible chunks; it also makes them more portable for using in future projects. Just grab a method and drop it into your next program and you’ve saved yourself a ton of time.

## Comment and comment well

Not only should you comment your code but you should also keep in mind a tip someone taught me: don’t just write what a section of code does, write why it is important. This helps to contextualize the code and presents the bigger picture of how this method or line fits into the grand scheme of things.

You can also use comments for a variety of other purposes. One trick I like is to use a kind of ‘keyword’ for code that needs looking at later, or code that I’m about to jump back to. If I need to quickly jump into another part of the code for reference, then using this keyword I can then perform a search to get back to where I just was. Likewise, if I earmark lines that need polish in this way, I can quickly sift through the page to find things that need brushing up.

avoid the temptation to simply comment out the code you no longer want

One last pointer: avoid the temptation to simply comment out the code you no longer want. This can be tempting as it allows you to save said code for later in case you need it, but it can hurt readability and make a project harder to navigate. If you’re anxious to delete old code, keep it in a notepad document or something instead.

JAVASELECT ALL

JAVASELECT ALL

//This is also a good place to write jokes for yourself, which will amuse/irritate you when you come back to

//look over your code.

## Don’t reinvent the wheel

The great thing about programming is that a lot of it is done for you. There are so many libraries, classes and example snippets of code that you’re free to use, that with some smart Googling you can pretty much build your app out of ready-made parts.

This saves a lot of time when building something complex. What’s more, is that if you’re liberating a piece of open source code from Github, chances are it has been worked on by multiple people and fine tuned to perfection. In other words, it’s probably better than the code you’d make if you quickly tried to piece something together yourself. You might even learn some good habits by looking at it.

Of course, it’s very important that you always give credit where it’s due and only use code with a Creative Commons license.

# Uncategorised

## Law of Demetor

The Law of Demeter (LoD) or principle of least knowledge is a design guideline for developing software, particularly object-oriented programs. In its general form, the LoD is a specific case of loose coupling. The guideline was proposed by Ian Holland at Northeastern University towards the end of 1987, and can be succinctly summarized in each of the following ways:[1]

Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.

Each unit should only talk to its friends; don't talk to strangers.

Only talk to your immediate friends.

The fundamental notion is that a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents), in accordance with the principle of "information hiding". It may be viewed as a corollary to the principle of least privilege, which dictates that a module possess only the information and resources necessary for its legitimate purpose.

It is so named for its origin in the Demeter Project, an adaptive programming and aspect-oriented programming effort. The project was named in honor of Demeter, “distribution-mother” and the Greek goddess of agriculture, to signify a bottom-up philosophy of programming which is also embodied in the law itself.

## High Cohesion

## Loose Coupling

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules.

## Modular

## Fluent Interface

Method for constructing object oriented APIs, where the readability of the source code is close to that of ordinary written prose.

## Encapsulation

Refers to the bundling of data with the methods that operate on that data. [5] Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them.

## Session

## Caching

## Logging

## Setting management

## Timing

## Object to Object Mapping (Auto Mapping)

## Domain Layer

### Entities

### Value Objects

### Repositories

### Domain Services

### Specifications

### Unit of Work

### Domain Events (EventBus)

### Data Filters

## Application Layer

### Application Services

### Data Transfer Objects

### Validating Data Transfer Objects

### Authorization

### Feature Management

### Audit Logging

## Distributed Service Layer

### ASP.NET Web API

### Web API Controllers

### Dynamic Web API Layer

### OData Integration

### Swagger UI Integration

## Presentation Layer

### ASP.NET MVC

### MVC Controllers

### MVC Views

### Handling Exceptions

### Localization

### Navigation

### Embedded Resources

### Javascript API

### CSRF/XSRF Protection

## Infrastructure Layer

### Background Services

### Background Jobs and Workers

### Hangfire Integration

### Quartz Integration

### Real Time Services

### Notification System

### SignalR Integration

### Object-Relational Mapping

### EntityFramework Integration

### EntityFramework Core Integration

### NHibernate Integration

### Dapper Integration NEW

# Left Out

## Resource acquisition is initialization (RAII)

# References

### <https://en.wikipedia.org/wiki/Abstract_factory_pattern>

### <http://www.oodesign.com/>

### <https://en.wikipedia.org/wiki/Software_design_pattern>

### <https://en.wikipedia.org/wiki/Architectural_pattern>

<https://en.wikipedia.org/wiki/Lean_software_development>

<http://deviq.com/repository-pattern/>

<http://deviq.com/strategy-design-pattern/>

<http://codebetter.com/jeremymiller/2005/09/13/inversion-of-control-with-the-plugin-pattern/>

<https://thenewstack.io/deployment-strategies/>

# Books

### <https://en.wikipedia.org/wiki/The_Pragmatic_Programmer>

### <https://en.wikipedia.org/wiki/Design_Patterns> by Gang of Four

### <https://en.wikipedia.org/wiki/Code_Complete>

### <https://martinfowler.com/books/eaa.html> by Martin Fowler

<https://en.wikipedia.org/wiki/Extreme_programming>

<https://itrevolution.com/book/the-phoenix-project/>

Other reading

<https://en.wikipedia.org/wiki/The_Mythical_Man-Month>