Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

On Patterns for Refactoring Legacy C++ Code into a Testable State Using Inversion of Control

by

Per Böhlin

LIU-IDA/LITH-EX-A--10/009--SE

2010-10-10

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

On Patterns for Refactoring Legacy C++ Code into a Testable State Using Inversion of Control

- Methodology, Implementation and Practices

Final Thesis

Master of Science in Computer Science and Engineering
at Linköping University
by

Per Böhlin

2010

Technical Supervisor: Patrik Höglund
Enea AB
Professor: Kristian Sandahl
Department of Computer and Information Science, Linköping University

Abstract

Amending old projects of legacy code to include agile practices such as extensive unit testing and refactoring has proven difficult. Since automated unit testing was not widely used a decade ago, much code has been written without unit testing in mind. This is especially true for C++ where RAII has been the dominant pattern .This has resulted in a lot of code that suffers from what best can be described as low testability. This also strongly impedes the creation of new unit tests to already existing code. Due to the lack of unit tests, refactoring is done sparsely and with great reluctance.

This thesis work tries to remedy that and, in the scope of a limited case study on an existing code base, looks into different ways of creating and utilizing object seams in legacy C++ code to decouple dependencies to make isolated testing possible. This regards to:

- What are the impediments for code to be testable in an isolated setting?
- What are the steps for refactoring code to a testable state?

The results can be summarized as to contain a list of factors affecting testability, among them: the use of asserts, global state, object instantiation, work in constructor and breaking Law of Demeter. Further with regards to patterns for refactoring code to a testable state, two types of patterns have crystallized: the injection of dependencies and the masking of dependencies using various techniques. The effect these two base patterns have on breaking dependencies on the base level and the Meta level is outlined. A catalogue of patterns has been compiled as part of the appendix.

Inversion of Control (IoC) is a principle used to decoupling classes and since strong dependences is often an attribute giving grievances with regard to testability, it was a central concern in this thesis. IoC can be simplified from a developer standpoint with the help of frameworks or what is usually referred to as IoC containers. Two IoC containers for C++ were evaluated:

- Autumn Framework
- PocoCapsule

In the evaluation of the two IoC containers it was concluded that Autumn was not mature enough as of the time of the evaluation to be used in production setting. PocoCapsule, even though compelling for some of its powerful features with regard to DSM and HOT, its configuration sometimes require in-code workarounds, affecting its usability in some set of scenarios.

However, the big difference was with regard to how the two containers approaches configuration. PocoCapsule uses static analysis of its XML configuration file making it truly declarative while Autumn does runtime parsing and dynamic invocations resulting in a situation closer to procedural scripting.

Acknowledgements

This thesis work could not have come about if had not been for Enea AB's continuous effort to explore current and future technologies. Their commitment to software quality through enhanced process and methodology has been an exemplary setting for learning new skills as well as a source of inspiration.

I want to extend a sincere *thank you* to the Enea Linköping office for their warm and openhearted welcome. The people here have gone beyond the call of courtesy and have made my time at Enea a most pleasant one. The fun and light-hearted atmosphere with the intelligent and knowledgeable colleges have been an excellent mix.

On a personal note, this project has been very rewarding, both in terms of new knowledge learnt and the hands-on experience gained. Even though the process has been meticulous, even exigent at times, it is the pinnacle of my academic life so far.

Linköping University, in particular the Department of Computer Science, should not go unnoticed. I have made a tremendous academic journey thanks to its experienced and dedicated staff.

Special thanks to:

Patrik Höglund, *technical supervisor at Enea AB*: even though not always physically present, thanks for all the support and valuable discussion throughout the project.

Prof. Kristian Sandahl, e*xaminer*: for the encouragement during difficult times and allowing me the flexibility needed.

Lattix Software, for providing me with a free extended academic license of Lattix LDM.

Programming Research Ltd, for providing me with an academic evaluation license of QAC++. A very well deserved thank you to the staff of PRL, most notably Justin Gardiner, who spent countless hours on emails and phone calls to make this happen.

SMACCHIA.COM S.A.R.L, for providing me with a professional version of CppDepend.

Per Böhlin

Linköping, September 4, 2010

Also thanks to:
Mikael Kalms, opponent
Mark Sin Deniz, and Torgny Hellström for reading and commenting on drafts.

Content

Tables, Figures & Code Examples

A. Tables

B. Figures

C. Code Examples

1. Introduction

1.1. Background

Test Driven Development (TDD), its origin often accredited to Kent Beck[1], is a software development practice that in recent years have seeped out from the Smalltalk and Java communities and has in many places become an accepted part of mainstream software development process. Robert Martin has gone as far as to claiming its practice is part of being a professional software developer [36]. TDD advocates test first development where simple unit tests are written prior to the code implementation that allows for the test to pass. This programming style results in that a lot of the production code is covered by tests.

The patron of this project, Enea AB, has experienced that many of today's projects originated prior to TDD and hence lack extensive unit tests, if any at all. Due to the deficiency of unit tests, refactoring is done sparsely and with great reluctance. Without the safety net of tests, change becomes tedious and difficult since it may inadvertently break other parts of the application. Hence, feature extension and maintenance risk becoming cumbersome in prolonged projects. Because of this, it has become apparent that legacy code, using Michael Feathers' definition[2], can benefit from being brought under test. [8]

Further, since much of the old code was written without unit testing in mind, it suffers from what can best be described as low testability, which strongly impedes the creation of new unit tests to already existing code. This furthers the barrier to switch to a TDD-style of developing.

Enea AB had an ongoing software development project matching the above scenario. In an effort to improve the state of that particular project as well as getting new general procedures for improving other projects of similar nature, Enea set up the project that in part resulted in this report.

1.2. Scope of Inquiry

A lot of code has been written under the device that objects should get their dependencies themselves. A way of doing this is to let the object create the dependency directly using stack or heap allocation. Another is to request the dependency from a centralized entity, usually implemented as a Singleton[3]. That results in tightly coupled code with strong dependencies that makes it hard to test in isolation. This due to the difficulty of stubbing/mocking[4] out those object dependencies; consequently preventing effective unit testing [39], [41].

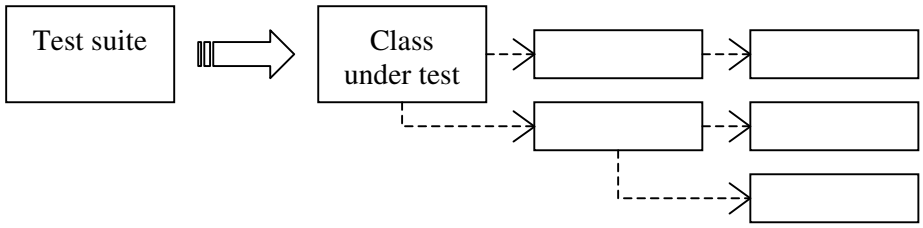


figure I. Dependency Graph

[1] [65], along with other resources (not listed here), accredit TDD to Kent Beck and his publication [1].

[2] Michael Feathers introduced a definition for legacy code as "code without tests" in [8] (preface).

[3] Singleton pattern from [11].

[4] *Stubbing/Mocking*: See section 1.5

Definitions for the distinction between stubs and mocks.

A way of writing more testable code is to use a form of *Inversion of Control* (IoC). IoC could be considered a general principle, at least according to Martin Fowler, where the control of execution is inverted and handed over to another component or framework [60], as in *figure II*.

However, in this report a more limited description will be used. In this context, IoC is the inversion of the control of configuration, creation and lifetime management. It is the question of inverting the control of configuration, creation and lifetime of objects from the objects that use them to an independent entity [52]; as in the form of an IoC container. This aspect of IoC is more interesting from a testability point of view as this allows for easier control of object graph composition and hence the possibility of replacing parts with test doubles.

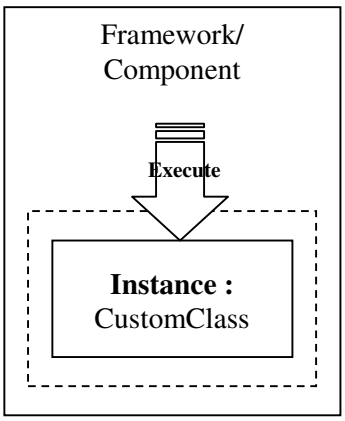


figure II. General IoC

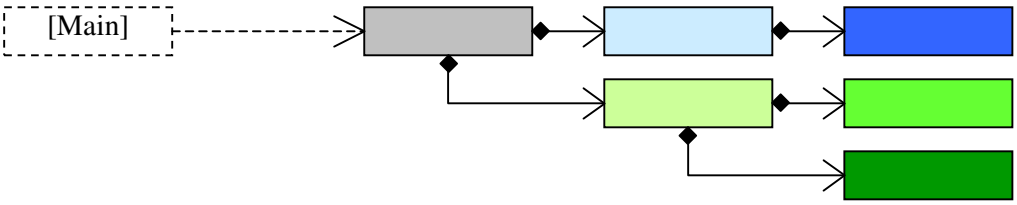


figure III. Traditional Object Ownership Graph

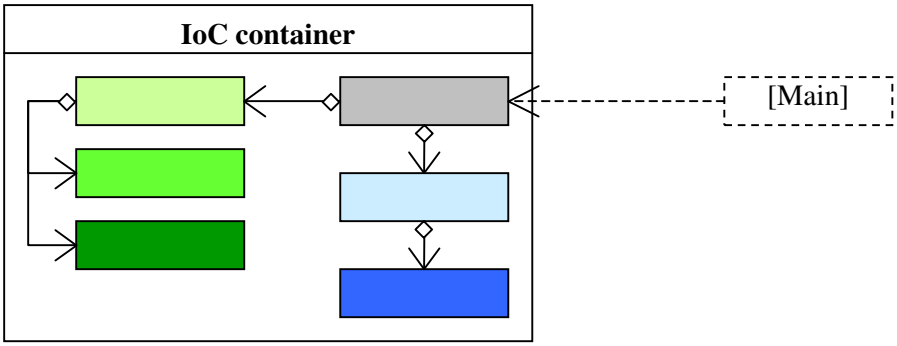


figure IV. Object Ownership Graph with IoC Container

Notice how the dependencies changes from being a multi level composite in figure III to a more loosely coupled graph of aggregates in figure IV. The IoC container has also been given the responsibility of creation and lifetime management denoted by the compositional association to the components represented by the colored boxes.

IoC has other benefits as well. By inverting the control of object creation and binding, the process of wiring an application together can be cleanly separated from the rest of the logic. For this to be feasible, components need to be cleanly separated with explicit interfaces and contracts. That is generally considered to be part of good design and hence IoC can be thought of as enforcing such a design. The loose coupling between components also makes it easy to extend behavior by introducing replacement parts. Since all wiring of components is done separately from the logic, reconfiguration of the application and introduction of new components can be done almost without risk.

One possible realization of IoC is through *dependency injection* (DI). This is a practice where dependencies are given to a class instance by injecting the depended upon object by passing a

reference to it to the constructor. Other methods are also available and a more thorough discussion can be found in section *3.8 Dependency Injection*. This allows for easy mock and stub substitution for systems under test. [39], [41]

Inversion of Control through dependency injection is one way of creating what is called a seam. Michael Feathers defines seams as "*a place where you can alter behavior in your program without editing in that place.*" [8](p. 31). This alternate behavior could be part of application configuration but what Feathers mainly refers to is alternate behavior during tests.

He further distinguishes between three types of seams [8](p. 29-44):

- **Preprocessor seams:** (static metaprogramming) altering behavior with the help of preprocessor macros such as `#define`, `#include`, `#if`, `#ifdef` statements for use with the C preprocessor.
- **Link seams:** replace functionality in the link stage, such as supplying an alternate but symbolically substitutable object file to the linker.
- **Object seams:** using polymorphism and subtyping to extend and modify behavior.

Object seams, where Dependency Injection belongs, are the focus of this report. Preprocessor and link seams are not considered.

The aim of the project is to improve the state of a codebase, denoted *Codebase I*, written in C++ for the Windows platform. It is a product maintained by Enea AB but owned and distributed by a company that will not be disclosed in this report. By introducing the concept of IoC through DI, and by general refactoring the code should become less tightly coupled and see an increase in readability and testability.

Furthermore, experience and knowledge should be gathered and result in guidelines, patterns if you will, for safe refactoring of legacy code to bring it to a testable state using the practices of IoC through DI. These patterns should cover a reasonable set of scenarios that can be encountered while refactoring and bringing legacy C++ code under test.

The intent is not for this collection of patterns to be complete, nor should it be seen as a comprehensive resource for general refactoring, not even from a C++ code perspective. Rather, the patterns cover a limited set of scenarios appearing with some regularity in the investigated codebase.

Testability, as a code attribute, will be a central consideration. Especially when refactoring code to bring it to a testable state. The aspects that constitute the concept of testability, will inevitably imbue and affect the scenario identification and pattern discovery. It will therefore be allowed to permeate large portions of the report as well as the results.

Inversion of control will be used as a means of isolating previously tightly coupled components/objects. Part of the scope of this investigative project is to evaluate IoC containers for use in conjunction with dependency injection. Containers selected for this are PocoCapsule (project website [97]) and Autumn Framework (project website [86]). The evaluation is not more comprehensive than to identify the basic effects on coding overhead, flexibility regarding component substitution and overall code readability. Due to licensing issues and the proprietary nature of the codebase provided by Enea, the IoC containers will be evaluated after use on a completely unrelated codebase denoted *Codebase II*. This is a simple 2D game in C++ written by the author a few years ago.

1.3. Objectives

Based on the scope of inquiry, stated above, the project's core objectives can be summarized in a set of research statements. These are intentionally stated in broad terms for conciseness, but should be read in the light of the limiting statements made in *1.2 Scope of Inquiry* and in *1.4 Project Directives and Limitations*.

The project can be generalized into three parts:

Part 1. Refactor selected classes in *Codebase I* so they can be brought under test.

Part 2. Identify common scenarios and patterns for refactoring that brings code to a testable state.

Part 3. Evaluate the use of IoC containers.

These objectives can be distilled further into three research questions that are the main focus of the investigation:

Q1. What are the impediments for code to be testable in an isolated setting?

Q2. What are the steps for refactoring code to a testable state?

Q3. What are the benefits and drawbacks of the two IoC containers evaluated, with respect to the second code base, Codebase II, under investigation?

1.4. Project Directives and Limitations

The project operated under the following specific directives and limitations, proposed by Enea AB. They apply to *Part 1* and *Part 2* of the investigation carried out on Codebase I. Except *D5,* which refers to Part 3 and involves Codebase II.

1.4.1. Directives

D1. Some code should be put under test as proof of concept and in part, increase the general code quality and hence benefit Enea AB.

D2. Refactored code should be loosely coupled and easy to read.

D3. Overall product quality should be maintained for Codebase I by regression testing following the same protocol as the general product.[5]

D4. The investigative part of the project should result in a set of patterns or strategies, covering common scenarios, for safe refactoring of legacy C++ code into a testable state, with adopted focus on the used codebase.

D5. Two IoC containers should be evaluated for use in terms of general benefit for projects of the same nature as the codebase used in this project. If licensing terms allows, incorporate the IoC container in the management of refactored components in the codebase provided by Enea.

1.4.2. Limitations

L1. General aspects of the application and the core module should be given focus. The graphics rendering module should not be considered due to incongruity to unit testing, nor modules that may contain privileged information or are classified in nature.

L2. Unit tests should be the focus when ever possible. Integration tests are not a requirement.

L3. Tests do not need to consider threaded environments.

1.5. Definitions

Aggregate: "*A class that represents the 'whole' in an aggregation relationship.*" [3].

Aggregation: "*A special form of association that specifies a whole-part relationship between the aggregate (the whole) and the component (the part).*" [3].

Association: "*A structural relationship that describes a set of links, in which a link is a connection among objects; the semantic relationship between two or more classifiers that involves the connections among their instances.*" [3].

[5] Enea AB's test protocol for the product is considered a trade secret and has no academic value for the scope of the report. No further description of it will therefore be given here.

Autumn: A C++ dependency injection framework. Project website [86].

Base level: Base level, or Meta level 0, refers to the runtime where software objects reside.

Bring under test: Bringing a class/method under test is the act of writing a unit test (if it does not already exist) and setting it up so that the class/method is tested satisfactory as part of an (automated) test procedure.

C++: General purpose programming language created in 1973 by Bjarne Stroustrup.

Class: "*A description of a set of objects that share the same attributes operations, relationships and semantics.*" [3]. Class is also used to denote a single class as part of object oriented design terminology, without taking its dependencies into consideration. Contrast with *Component*.

CLR: Common Language Runtime. Microsoft's implementation of the runtime environment adhering to the Common Language Infrastructure (CLI) standard. CLR operates on Common Intermediate Language (CIL) code and just-in-time compile it to run natively on the underlying platform. It serves similar purpose to the JVM and Java ByteCode.

Codebase: "*whole collection of source code used to build a particular application or component*." [63]. In this report, codebase refers to the specific codebase investigated in the project, if not stated otherwise.

Component fragment: A class that lacks semantic value on its own. It can be thought of as an aggregate or composite without its dependencies. The terminology is used to emphasize that the class is incomplete without its dependencies.

Component: "*A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.*" [3]. In the context of this report, component refers to a software source code component. It is used interchangeable to denote classes (that lacks dependencies), aggregates and composites; any piece of code that can be thought of as a unit and allows reuse. This extension is a slight maltreatment of the strict definition that requires that the component adhere to a specified service interface.

Composite: "*A class that is related to one or more classes by a composition relationship.*" [3].

Composition: "*A form of aggregation with strong ownership and coincident lifetime of the parts by the whole; parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it; such parts can also be explicitly removed before the death of the composite.*" [3].

CORBA: Common Object Request Broker Architecture. A standard defined by the Object Management Group (OMG) that is cross language, cross host and cross platform.

CUT (Component/Class Under Test): Refers to the component/class being tested in a unit test scenario.

Dependency: In [3], dependency is defined as "*A semantic relationship between two things in which a change to one thing may affect the semantics of the other thing.*" In this report, the term is expanded to encompass syntactical relationship as well. This since the exclusions of one thing may prevent compilation (syntactical) but do not affect a limited operation such as a test case. Hence, any entity that prevents compilation of a class by its inclusion or absence is considered a dependency.

DI: Dependency Injection.

DOC: Depended On Component/Class.

DSM: Dependency Structure Matrix. A matrix showing dependencies between individual classes and components. The matrix consists of a row and a column for each class. If the cell where the row for class A intersects with the column for class B has the value of 4, it means that A depends on B with the quantity of 4, in the used metric. Different metrics can be applied. If efferent coupling is used as the metric, the value of 4 would mean that class A calls 4 of B's methods.

DSM: Domain Specific Modeling.

EI: Extract Interface, a pattern for refactoring. See Appendix A for description of the pattern.

ELMI: Extract Local Minimized Interface, a pattern for refactoring. See Appendix A for description of the pattern.

Fake object: An object that imitates the behavior of another object and is used together with CUT, in order for the CUT to be tested in isolation. In this report it is used interchangeable with *Test double*. See also *Mock object* and S*tub*.

Friendly: An object that can be controlled in a test situation. It can be a fake object, but also a real implementation which has been tested in isolation and can be configured in a way that makes sense for the test setting at hand.

Getter: A method named `getPropertyName` and is used for retrieving the value of a property.

IDE: Integrated Development Environment.

Injection Path: Collective name for different explicit ways of injecting a component into another, different types of DI, such as constructor injection, setter injection, etc.

Integration test: Tests several components together. In contrast to *unit tests*, it is not uncommon for integration tests to access disk, databases and network recourses.

Interface: "*A collection of operations that are used to specify a service of a class or a component.*" [3]

IoC container: Inversion of Control container, a container which holds a completed application by wiring components together. The schema for the wiring is often defined in a neutral language such as XML. The IoC container also manages the lifetime of the components it creates.

IoC: Inversion of control.

JVM: Java Virtual Machine. The program and the set of libraries adhering to the Java API, required to operate on the intermediate language code known as Java bytecode. It serves similar purpose to CLR and CIL code.

LGPL: GNU Lesser General Public License, see [91]

Link seam: Replace functionality in the link stage, such as supplying an alternate but symbolically substitutable object file to the linker. See also *seam*.

LMI: Locally Minimized Interface, a pattern for refactoring. See Appendix A for description of the pattern.

Meta level: In this report Meta level refers to what in other contexts might be described as Meta level 1. Meta-objects, such as classes reside here and describe elements on the base level.

MI: Minimized Interface, a pattern for refactoring. See Appendix A for description of the pattern.

Mock object: a *fake object* which in addition to being a stand-in for a real object, it also records operations that are invoked on the object so that they can be asserted that they actually took place during the test. Compare to *Stub*.

Null Object: Also referred to as Null Object Pattern. An object that adheres to a syntactic contract of a type but does not have any actual implementation and does no real work. It is a substitute for null where null checks should be avoided.

Object seam: Using polymorphism and subtyping to extend and modify behavior. See also *seam*.

Object: "*A concrete manifestation of an abstraction: an entity with a well-defined boundary and identity that encapsulates state and behavior: an instance of a class.*" [3]

OO: Object Oriented.

OOD: Object Oriented Design.

OOP: Object Oriented Programming.

PocoCapsule: An IoC container and DSM (Domain Specific Modeling) framework for C/C++. Project website [97]

Preprocessor seam: (Static metaprogramming) Altering behavior with the help of preprocessor macros such as `#define`, `#include`, `#if`, `#ifdef` statements for use with the C preprocessor. See also *seam*.

Refactoring (noun): "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*". [9] (p. 53)

Refactoring (verb): "*to restructure software by applying a series of refactorings without changing its observable behavior*". [9] (p. 54)

Refactoring, complete: Complete Refactoring leaves the subject with no obvious imperfections. The subject abides to the heuristics of what can be considered good design and is in harmony with the system given the systems current design.

Refactoring, deep: Deep Refactoring affects dependencies on the Meta level and therefore also on the base level.

Refactoring, partial: Partial refactoring leaves the subject in an imperfect state. There are obvious improvements still to be made but dependencies to other code parts prevent it.

Refactoring: shallow: In contrast to deep refactoring, affects shallow refactoring only dependencies on the base level.

Reflection: A programs ability to observe and modify its own structure and behavior. This is done by giving access to Meta objects such as class definitions (Meta level) in runtime (base level).

SCA: Service Component Architecture. A specification for Service Oriented Architecture.

Seam: "*A seam is a place where you can alter behavior in your program without editing in that place.*" [8](p. 31)

Setter: A method named `setPropertyName` and is used for setting a property or injecting a dependency.

SGML (Standard Generalized Markup Language): "*an ISO-standard technology for defining generalized markup languages for documents*" [66]

SOAP: Simple Object Access Protocol. A protocol for structured information exchange with Web Services. SOAP is protocol layer on top of Remote Procedure Call (RPC) and HTTP.

Stub object: A *fake object* that acts as a stand-in for a real object in an isolated test scenario. The stub object can serve a limited set of expected data required in the test scenario. In contrast to a *mock object*, a stub does not record events and are hence not asserted against.

SUT (System Under Test): refers to a component or a set of components that interact, being tested.

TDD: Test Driven Development.

Test double: In the context of this report, same as *Fake object*.

Unit test: Simple test designed to test a small unit of code that the tester controls such as a method, function or class. The entire test takes place in memory, does not access disk, databases or network resources and should run in milliseconds.

VS: Microsoft® Visual Studio.

WSDL: Web Service Description Language. A definition language to describe Web Services.

XML (eXtensible Markup Language): a markup language, extended from SGML, developed by World Wide Web Consortium [104].

1.6. Report Disposition

Chapter *2 Methodology* and chapter *3 Theory* describe the general methodology used and the theories relied on to reach the outcome presented in chapter 7 *Results*.

Chapter *4 Case Study*, describes the actual work done.

Chapter *5 Tools and IDE used* complements *Methodology*, *Theory* and *Case Study* by providing an outline of the technology environment set up, which tools were considered, which were available and used, their rationale and application.

Chapter *6 Status of the Codebase at Start* describes the state of Codebase I when the project started. This is used for contrasting the results and gives an insight in the progression accomplished in the form of how the code has improved during the thesis work.

Results presents the outcome of the work described in chapter *4 Case Study*, after applying the methodology and theory described in previous chapters. The chapter contains information on the status of Codebase I after refactoring as well as a discussion of the patterns and scenarios discovered. Complete description of the patterns and scenarios can be found in *Appendix A*.

Chapter *8 Conclusions* discusses the general conclusions regarding the results as well as the project's overall success rate.

Future work points at areas of further investigation and other related technologies that are of interest.

1.7. Intended audience and reading suggestions

This report is written with software developers in mind. To readers without at least basic understanding of C++, software design and practices, this thesis will seem inaccessible.

If the academic dimension of the report is of no interest, then chapter *2 Methodology* can be skipped.

Readers knowledgeable in Test Driven Development, Refactoring, Unit Testing, Testability, Inversion of Control and Dependency Injection can pass over chapter *3 Theory*.

Readers not interested in the scientific method but only in the result can skip to chapter *7 Results*, and Appendix A.

1.8. Appendix Overview

Appendix A. Contains a list and full descriptions of all patterns and scenarios identified. It can be considered a handbook, of detailed steps on how to safely refactor legacy C++ code to bring it to a testable state.

Appendix B. Contains a summary of the different dependency injection techniques presented in this report.

Appendix C. Contains the full definition-file, XML, used when evaluating PocoCapsule.

Appendix D. Contains the full definition-file, XML, used when evaluating Autumn Framework.

Appendix E. Contains full source code example of manual wiring of components using dependency injection

1.9. Third Party Libraries and Applications

A number of different third party libraries and tools where used in the project:

CppUnit (project website [101]), a C++ port of the widely used Java unit testing framework JUnit. All unit-testing was done in CppUnit.

Lattix LDM, from Lattix Software(company website [94]), was used to generate Dependency Structure Matrixes (DSM).

BullseyeCoverage, from Bullseye Testing Technology (company website [87]), was used for test coverage.

QA C++, from Programming Research Ltd (company website [99]), is a static code analysis tool used in this project for gathering code quality metrics.

CppDepend (product webpage [90]) was also used for static code analysis with the purpose of collecting metrics.

For a complete list of the technologies used, see chapter *5 Tools and IDE used*.

1.10. Proprietary Information

The codebase provided by Enea AB used in the project is strictly proprietary in nature, and may even contain classified information. Details about it, such as product name, have therefore intentionally been left out. Throughout the report, the product code is referred to as *Codebase I* or *code under investigation*.

Code examples will be given in a very narrow context and partially obfuscated in order to comply with non-disclosure agreements, copyrights and confidentiality agreements. Code published in this report and its appendices should not be considered a license of use, nor is it consent to reproduce said code. All rights are reserved by the copyright holder.

1.11. Coding Conventions in Examples

A number of code examples are given throughout the report. A list of all examples can be found on page *vi*. The examples show the following properties:

- All code examples are given in C++ except IoC container configurations that are given in respective container configuration language.
- Class member variables are denoted by the `m_` prefix.
- `#define` constants are written in all capitals.
- Classes that start with `I`, as in `IDatabase`, denote a pure virtual class (interface).
- All code examples are written in boxes as the one below.

Code example X. Title

```
{

}
```

1.12. Conventions for References

References are given by [x] where x refers to the number in the reference list. [x] listed inside a sentence only apply to that sentence. [x] at the end of a paragraph, applies to the entire paragraph.

If relevant, such in the case of quotations, page numbers are given in the form [x] (p. xx) when applicable. In the case of video or audio, time is given as [x] (time: 00:00) where first set of digits are in minutes and the second set in seconds.

2. Methodology

In this chapter the general research methodology is outlined. The purpose and limitations of certain techniques are addressed as well as their origins and scientific value. This chapter is safe to skip for readers who are not interested in the academic dimension of the report.

2.1. Scientific View

Scientific rigor originates in its ability to reproduce objective results. Further, one could argue that a strict positivistic view is necessary in order to achieve said objectivity. However, the discipline of software development is not an exact science in that respect. It is both a craft and an engineering discipline [4], [13], [24], [31], [32], [35], [36], [61] and a less strict view is consequently adopted for a meaningful evaluation of progress should be possible.

Therefore, an inductive view is assumed, that does not suffer from the constraints of positivism and scientism. The project is for all intended purposes an inductive case study[6]. All parts of this project's research areas should be considered intensive normative [72]. Even though *Part 2*, *Identify common scenarios and patterns for refactoring*, can be said to have the ambition of extensiveness, in reality, due to the limited sample and scope, it by all rights must be considered intensive.

For the purpose of this report an ontological postulation of the existence of a single objective reality, independent of observers' subjective conscience is made. An epistemological postulation of that meaningful knowledge about said reality can be obtained through observation is further made. Hence we adopt the criteria required for a knowledgeable discussion within the scientific view presented above [7].

2.2. Literature Study

As seen in the reference list, a number of sources were consulted; books, articles, blogs, audio and video recordings of talks given by prominent people. So in this context literature is more than the printed word. It refers to any published resource. A common problem with the entire field of software engineering is that very little of the accepted knowledge derives from controlled scientific experiments, often not even based on empirical data. Many of the so called best practices, patterns and expert advice originate instead in many years of collective field experience.

This of course constitutes a problem, not only for the scientific value of this report, but for the software industry as a whole.

That being said, the sources relied upon are mainly attributed to people who are recognized in the software community as either authors of what can be considered best sellers in the software development world, prominent and often invited speakers and/or experienced and successful consultants.

Due to the topics addressed, that of refactoring, test driven development and unit testing, a skew toward authors of the agile mindset can be noted.

2.3. Investigative Method

As recognized in earlier sections, software development is a craft, or trade as well as an engineering discipline. The available literature and general direction of the field further encouraged such an outset.

At the time of the project, neither a suitable model of investigation nor authoritative Meta method could be found. That led to a very practical approach. The codebase under investigation was refactored using techniques discussed in referenced resources or discovered first hand. The process can be very

[6] Even though the deficiencies of inductive reasoning, as outlined by Chalmers in [5] are recognized, the general acceptance for inductive methodology makes it a viable method in this context [7].

much characterized as a trial and error effort. Code was refactored using a set of techniques leading it down one path, only to recognize failure, revert and try again.

This heuristic approach shares similarities to the one used by Kent Beck when creating his own set of patterns and principles which later would be published in *Smalltalk Best Practice Patterns* [62] (time: 0:00 – 4:00 min).

Some of the most widely used resources by the everyday developer are among others, Beck's [1], Fowler's [9], Feather's [8] and Martin's [24] –all prominent and major contributors to the software development best practice field– all lack scientific rigor but rather follow similar heuristic methods used here.

In [12], Goldkuhl discusses the possibility of a Practice Science perspective for Information Systems (IS). He further characterizes IS as an artifact that humans both design and use. In a similar way, object oriented code is designed and used. The design answers the questions of what classes should there be, how they interact, what their interfaces and dependencies are. Programmers then use those classes to construct complete systems/applications. The similarities between IS and programming in regard to the design and use just described, as well as the proximities of the two fields, makes it reasonable to apply Goldkuhl's Practice Science perspective on the software development field as well.

The lack of reputable meta-methods is of course a weakness, from a stringent scientific perspective. However, the overall result should, due to the other factors stated above, hold some value even if its limitations should be recognized.

In an aspiration to balance these shortcomings, this report contains a quantitative evaluation of the change in code quality –discussed in chapter *2.4 Quality Metrics*.

2.4. Quality Metrics

A set of code quality metrics will be used as a way of validating the value of the testability impediments and refactoring patterns gathered. The aim is to see if there are any difference between the metrics of testable code and non-testable code. This is intended to support the claim that better code quality has been achieved with the refactoring methods suggested.

The choice of metrics was based on two criteria. The metric should be:

- a valuable measure for testability, good OOD, or readability
- measurable using an automated toolset available for MS Windows

The second criterion comes from the need to analyze a code set large enough to make it unfeasible doing so by hand. In addition, the toolset should fit with the rest of the tools used in the project. Integration and cross-platform compatibility was a problem domain outside the scope of the project. An implicit criterion was that the tool/application must be available, in concern to licensing, at the time of the project. Hence, some metrics had to be left out.

One such metric was the Testability Score suggested by Misko Hevery where a class is given a score based on its ability to be tested in isolation [64]. Unfortunately, Testability Explorer (Project website [89]), the tool used to calculate the score was at the time of writing only available for Java code.

In the rest of this section, and its subsections, a collection of code quality metrics will be explained and motivation given to why they have been used.

2.4.1. Abstractness

Dispite the initial goal of only collecting metrics using automated tools, the Abstractness metric was obtained through manual count. This exception was made since the metric is easy to collect manually and it is very important with regard to object-oriented design. The metric is discussed in a broader context in section *3.6 Dependency/Coupling*.

$$A : Abstractness : \frac{number\ of\ \ abstract\ methods}{total\ \ number\ of\ \ methods}$$

- Pure virtual methods are counted as abstract
- Constructors are not counted at all since they have special meaning and cannot be virtual.
- Virtual destructors are considered abstract in this context. Non-virtual destructors count as non-abstract methods.
- (Static) class methods count as non-abstract.

This metric is a variation on the abstractness metric proposed by Robert C. Martin in [18].

2.4.2. Association Between Classes

"*The Association Between Classes* (ABC) *metric for a particular class or structure is the number of members of others types it directly uses in the body of its methods.*" [29]

The metric gives an indication on how coupled a class or structure is to its surroundings. A high value indicates strong coupling and might prevent reuse and make the class difficult to test in isolation. It could also suggest too many responsibilities.

2.4.3. Coupling Between Objects

Coupling between objects (CBO) measure the number of methods (member functions) or member objects of other classes accessed by a class [6]. This count does not include classes within the inheritance tree. A high count suggests strong dependency on other objects and makes it more difficult to test the class in isolation. This metric is one of Chidamber's & Kemerer's original measures for object oriented design quality.

2.4.4. Coupling

C_a: Afferent coupling, inbound coupling.

"*The Afferent Coupling for a particular type is the number of types that depends directly on it.*" [29]

C_e: Efferent coupling, outbound coupling.

"*The Efferent Coupling for a particular type is the number of types it directly depends on. Notice that types declared in framework assemblies are taken into account.*" [29]

See section *3.6 Dependency/Coupling* for discussion of the metric.

2.4.5. Cyclomatic Complexity

Cyclomatic Complexity can be counted in many different ways and applied on many different levels.

CppDepend claim to calculate it as:

1 + {the number of following expressions found in the body of a method}

if, while, for, foreach, case, default, continue, goto, &&, ||, catch, ternary operator ?:, ?? [29]

However, experience show that it does not use the plus one.

CC_{tot} refers to total cyclomatic complexity for a type/class using CppDepend's method.

CC_{max} is the highest cyclomatic complexity for any method of a type/class using CppDepend's method.

CC_{ave} is the average cyclomatic complexity of a type's/class' methods using CppDepend's method.

QA C++ calculates cyclomatic complexity as the number of decisions plus 1. It does not take ternary operator ? : into account. Neither is logical operators such as && and || considered.

CYC_{max} is the highest cyclomatic complexity for any method of a type/class using QA C++ method.

$CYC_{\max} = \max(CYC(m_i), m_i \in \mathrm{M}(c), i = \{0,...,n\})$, where $\mathrm{M}(c)$ is a set of class c's methods.

$CC_{\max} = \max(CC(m_i), m_i \in \mathrm{M}(c), i = \{0,...,n\})$, where $\mathrm{M}(c)$ is a set of class c's methods.

$CC_{ave} = \frac{CC_{tot}}{\#m+\#sm}$, where #m is number of instance methods, #sm is number of class methods.

2.4.6. Depth in inheritance tree

From Chidamber's & Kemerer's metric suite:

"*Depth in inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.*" [6]

A high number of ancestor classes might make a class difficult to understand.

2.4.7. Instability

Robert C. Martin's OOD metric from [18].

I: Instability: $\frac{C_e}{C_a + C_e}$

The metric was calculated from C_e and C_a acquired using CppDepend. See section *3.6 Dependency/Coupling* for discussion of the metric.

2.4.8. Lack of Cohesion of Methods

Lack of Cohesion of Methods (LCOM) is a metric that measures the lack of cohesion in methods for a class. Since cohesion is an important concept in OOP the metric can be used to measure the state of the code from an OOD perspective. A number of different models exist. LCOM was proposed as an OOD-metric by Chidamber and Kemerer in [74] and later revised in [6]. The exact formula for LCOM has however changed since then and now exists in several different versions. The version used here will be determined by the tools used.

LCOM: Lack of cohesion of methods of a class (CppDepend)

LCOM (HS): Lack of cohesion of methods of a class using Henderson-Sellers formula

CppDepend defines:

*LCOM = 1 – (sum(MF)/M*F)*

LCOM HS = (M – sum(MF)/F)(M-1)

Where:

M is the number of methods in class (both static and instance methods are counted, it includes also constructors, properties getters/setters, events add/remove methods).

F is the number of instance fields in the class.

MF is the number of methods of the class accessing a particular instance field.

Sum(MF) is the sum of MF over all instance fields of the class. [29]

LCOM has a range of [0-1] and LCOM (HS) a range of [0-2]. LCOM=0, LCOM (HS)=0 indicate complete cohesiveness.

LCM: Lack of Cohesion of Methods within a class (QAC++):

QAC++ uses a variation of Chidamber's and Kemerer's original definition from [6]. QAC++ uses a formula that could be defined as:

Consider a Class C_1 with n methods M_1, M_2, .., M_n. Let $\{I_j\}$ = set of instance variables used by method M_i. There are n such sets$\{I_1\}$, …, $\{I_n\}$. Let $LCM = \left|\{(I_i, I_j) \mid I_i \cap I_j \neq \theta\}\right|$

2.4.9. Lines of code

Lines of code (LOC) is a measure of program size. When applied to individual classes and methods, it can be a measure on readability. Small functions are usually easier to read. A high value for a class could indicate that the class should be split into several smaller classes. A high value for a method could indicate a need to extract parts of it into a new method or push to a sub- or superclass depending on cohesion and inheritance structure [8].

The threshold for what constitutes too big a method is very much individual taste. Martin declares for only a few lines of code per function in [24]. A general rule of thumb often heard is that the function should at least fit on the screen. This rule has become less and less viable as screen size increases and it is becoming more and more common to replace the fit to screen measurement with 20-25 lines. QA C++ recommends not exceeding 200 lines of code.

In this report, the following related metrics are used:

- LOC: Lines of code for an entire class.
- LOCM: Lines of code of method.

$LOCM_{max}$ indicates the highest value found among a class' methods. $LOCM_{ave}$ is the average lines of code per method.

$LOCM_{\max} = \max(LOCM(m_i), m_i \in \mathrm{M}(c), i = \{0,...,n\})$, where $\mathrm{M}(c)$ is a set of class c's methods.

$LOCM_{ave} = \frac{LOC}{\#m + \#sm}$, where #m is number of instance methods, #sm is number of class methods.

2.4.10. Maximum nesting depth of control structures in a method

Maximum nesting depth of control structures in a method (ND_{max}) gives a measure of how complex a method is. Note that the nesting level can sometimes be difficult to spot. Structures of "else if" are usually written with the same indentation even though they increase the nesting depth.

2.4.11. Number of base classes in inheritance tree

The number of base classes in inheritance tree (#BC) gives an indication of the complexity of a class. A class with many ancestor classes might be difficult to understand. #BC differs from NOP in that it takes all ancestor classes into account, not only the immediate parents. It also differs from DIP since DIP only counts the deepest path through the inheritance tree, while #BC counts all ancestor classes. In case of single inheritance DIP and #BC will have the same value.

2.4.12. Number of Fields in Class

The number of fields (#fld) of member variables a class has. If a class has many member variables, together with high LCOM, it is usually a sign of the class doing too much and should be split into smaller classes.

2.4.13. Number of immediate children

The number of immediate children (NOC) gives an indication of how complex the inheritance hierarchy is. If many classes have high values for NOC, it could indicate a complex design. In addition, if no classes have children it could indicate code duplication and lack of reuse. Classes with high values for NOC are highly dependent and should therefore be abstract (high abstractness) and stable

(low instability). NOC also gives an indication on how fast an inheritance hierarchy fan out. The metric is part of the CK (Chidamber & Kemerer) Metric Suite [6].

2.4.14. Number of immediate parents

Number of immediate parents (NOP) is the number of immediate parent classes a class has. Root base classes have a value of 0 and single inheritance results in a value of 1. Values higher than one could indicate complex structures. Multiple inheritance is usually discouraged unless done using mixins.

2.4.15. Number of Instance Methods in Class

The number of instance methods in a class (#M). High numbers might indicate that a class is doing to much and should be split into smaller classes, especially if LCOM is high as well. Usually when talking about methods, instance methods are referred.

2.4.16. Number of Static Methods in Class

The number of Static methods in Class (#SM) -also referred to as class methods. Class methods should generally be avoided since they cannot be made virtual and overridden. Programs with high #SM can be procedural in nature and therefore suffer from poor testability.

2.4.17. Number of subclasses

Number of subclasses (#SC) differ from NOC in that #SC takes all child classes into account, not just the immediate classes. While NOC gives an indication of how fast an inheritance hierarchy fan out, #SC gives a better value for total complexity. Classes with high values of #SC should be abstract (high abstractness) and stable (low instability). Large differences in NOC and #SC can indicate deep inheritance structures that are overly complex.

2.4.18. Response for Class

Response for Class (RFC) is the cardinality of a class' response set. That is the number of methods on a class plus the number of unique methods on other classes, and unique functions, called by a class' methods.

If class C_1 has two methods M_1, M_2. Class C_1 has a member variable of type C_2. M_1 calls M_a and M_b on C_2. M_2 calls M_a and M_c on C_2. M_2 also calls stand-alone function F_1. This results in a RFC for C_1 of 5 (2 +3). M_b is only counted once even though it is called from two different places.

A large value for RFC usually indicates that a class will be difficult to test.

2.4.19. Static Path Count

The static path count (SPC) is an estimation of the *true* number of paths. Each condition does not consider other conditions but is disjoint. The true path count can therefore be lower than SPC.

The following condition is most often true:

Cyclomatic complexity ≤ true number of paths ≤ Estimated static path count

SPC_{max} is the maximum of a class' methods SPC.

$SPC_{\max} = \max(SPC(m_i), m_i \in \mathrm{M}(c), i = \{0,...,n\})$, where $\mathrm{M}(c)$ is a set of class c's methods.

2.4.20. Type rank

Type rank is a dependency measure. It is computed by applying the Google PageRank algorithm on the graph of types' dependencies. In the case of the CppDepend implementation, a homothety of center 0.15 is applied to get an average TypeRank of 1. [29]

2.4.21. Weighted Methods per Class

Weighted Methods per Class (WMC) is part of the CK (Chidamber & Kemerer) Metric Suite [6]. It is the sum of the cyclomatic complexity of all the methods in a class. High values for WMC indicates complexity.

This metric is measured using QA C++ and therefore uses there definition of cyclomatic complexity (CYC). QA C++ definition of CYC is at least one (1 + number of decisions), while CppDepend's version (CC) has lowest value of zero. Therefore, CC_{tot} will be different from WMC.

2.5. Profiling Tools

A number of profiling tools was used to measure tests code coverage and code quality metrics. The lack of free tools for C++ on the Windows platform soon became apparent. Therefore, the tools finally used were only done so at the end of the project for the purpose of measuring the final state. Hence, the real benefit of such tools, the ongoing progress, was not utilized.

Even good commercial tools were difficult to come by. The list finally used, only came about after extensive search and negotiation with involved companies. Other tools may be available but was not used because they were not available at the time of the project, free academic licenses would not be granted or the product was simply not found in time.

3. Theory

This chapter presents some of the theories and current knowledge of TDD, Testability, Unit Testing, Refactoring and OOD. The discussion is not complete. These topics are much too vast to all be handled properly in the scope of this report. The purpose is to lay the ground work for the discussion featured in *7 Results*. The topics are only explained and nuanced enough to give relevance to the results later presented.

It is safe to skip this chapter for readers who are well versed in Extreme Programming and its practices or other practices subscribing to the agile way.

3.1. Previous Work and Current Methods

The software development field has only existed for some 50 years and therefore suffers from some of the problems expected of such a relatively young discipline. In addition, software development has a very low barrier of entry. Anyone can pick up a compiler and start programming. Many people, who are experts in other areas, have come into programming to solve specific problems related to their original knowledge-domain.

As a result, software development is a field of many practitioners without formal training. The practices deployed are in many cases based on general experience rather than scientifically verified knowledge or empirical evidence.

Even formal publications for field journals and alike often lack the scientific rigor required in other disciplines. According to a study performed at Blekinge Institute of Technology, as much as 81% of the publications reviewed in the study contained no comments on or only unverified statements on application/validation of the results of the solutions proposed in the reviewed publications [15].

One possible reason for this heuristic approach is the lack of clear distinction between engineering discipline and craftsmanship/artistry. As stated in previous sections, software development is both an engineering discipline as well as a craft. This makes it very difficult to study, even more so to produce verifiable objective results in the strict scientific sense.

That said; a lot has been accomplished in the last decade. The software development community has seen an upswing in best practices. In the community summarized as agile[69] (practicing agile methodologies such as Extreme Programming, Scrum, Lean), practices such as test driven development, refactoring and continuous integration together with a stronger sense of craftsmanship has laid the ground for increased professionalism and demand for quality. Quality here is stated quite broadly and is done so purposely. In this context it refers to the quality of development methods, day to day practices, tools used and code quality in respect to lack of defects, simplicity, maintainability, readability, testability, structure, among other things. [24]

In the following sections some of the established practices are explained. It should not be seen as a complete description of those practices or a formal argument of their pros and cons. It is merely a brief account to explain these practices as they relate to the topic of this report.

3.2. Test Driven Development

Test Driven Development, TDD, or Test Driven Design as it is sometimes referred to, is not by itself relevant for the research questions explored in this project. However, for a more complete understanding of testability, refactoring and unit testing, one has to understand its relations to TDD as it is fundamental for the agile mindset.

Test Driven Development employs what is referred to as Test First Programming [1]. That is the practice of writing the test before the code. In [2], Kent Beck outline the following development cycle:

- Add a new test.
- Run all tests and see if any tests fail.

- If it does, write some code –if not, start over and add another test.
- Run all tests again to verify that the added code makes the test pass.
- Remove duplication, clean up the added code and verify by running the tests again.
- Start over.

The development cycle can also be expressed more compactly as:

- Create a failing test
- Write just enough code to pass the test
- Refactor your code to improve design

This cycle is also known as Red/Green/Refactor. The color reference is to how some unit testing framework's IDE plug-ins display failing and passing tests with a red and green bar, respectively [2] (p. x).

The tests should be automated and fast since they are to be run very often. Further discussion on test design can be found in section *3.5 Unit Testing*.

According to Beck, this practice addresses many problems at once [1] (p. 50). The two most important ones are perhaps:

- **Scope creep:** Less risk of writing code "*just in case*". The code written becomes more focused and has explicit intent.
- **Coupling and cohesion:** Hard to test code, due to extensive coupling to other segments or lack of cohesion, are often an indication of an inherent design problem.

Code written with test first programming has, if done right, a high quality with regard to design and as a corollary – high testability. See section *3.3* for a continued discussion of the attribute of testability.

A misconception about TDD is that it is all about testing and quality assurance –it is not. TDD is about designing software and documentation. Martin puts it this way:

"The act of writing a unit test is more an act of design than of verification. It is also more an act of documentation than verification. The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function."[23] (p. 23)

Scott Bellware, Agile coach and consultant, dedicates an entire show on HanselMinutes to the point that "*TDD is more about design than about testing*" [33].

3.3. Testability

Traditionally, easy to test code is often thought of as having low cyclomatic complexity and few static code paths. Those are of course important measures for testability, but not the primary ones when it comes to testability in the agile sense. When referring to testability, it really refers to easiness to test code [33], mostly in terms of coupling to other code and internal cohesion.

Code that:

- has few dependencies,
- has explicit dependencies,
- is simple,
- and has high cohesion

is generally easy to test. This since few and explicit dependencies are generally easy to replace with *friendlies*. This is the most important aspect of testability, that a class can be tested in reasonable isolation without massive configuration and pre-set state.

Classes with high cohesion usually have only a few responsibilities –preferably one. Added with simplicity in the form of low cyclomatic complexity and few static code paths, you generally have a class with few methods which use only can result in a limited set of state variations.

Testability also correlates to high code coverage. 100% code coverage by unit tests should be the goal for all written code –user interface code exempted. If high numbers are difficult to obtain, that in itself is usually a sign of poor testability [33].

3.4. Refactoring

Martin Fowler defines refactoring as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*". [9](p 53). Robert C. Martin uses a similar definition: "*the act of changing the structure of code without changing its behavior as proven by executing a suite of tests*" [58](44:08 min).

A consequence of Martin's definition –and to some extent Fowler's, depending on your interpretation– is that you can not do refactoring without tests in place. Hence you have a problem when code cannot be brought under test without refactoring.

To escape this quandary, a more moderate definition is used for the context of this report. Here refactoring denotes "*a change made to code to make it easier to understand and cheaper to maintain, with the intent not to change the implicit behavior and feature set of that code*".

That should be understood as to allow for local as well as global changes in code syntax to make a class testable, as long as it does not alters the existing behavior of that class. This more relaxed definition allows for the extraction of interfaces, encapsulation of public member variables, among other things, to fall within the stipulated provisions of refactoring.

Due to the less strict definition of refactoring used in this report, a distinction in scope is necessary. See the discussion on internal and external refactoring later in this section.

The main purpose of refactoring is to clean up the code after the first write. Code is rarely as good as it can be the first time you write it. Further on, change of direction forces code to change over time and sometimes makes it obsolete. Being afraid of changing or deleting existing working code will eventually lead to disaster. Code requires maintenance, otherwise it will eventually rot. Refactorings offer structural patterns for changing code in a predictable and relatively safe manner.[9] (p. 53-57)

Murphy-Hill and Black distinguish between two types of refactoring:

"*You perform floss refactoring to maintain healthy code, and you perform root canal refactoring to correct unhealthy code*" [26] (p. 7)

The former is characterized by small but frequent changes. The latter is done as an invasive action in an attempt to salvage code in a deteriorated state. To continue the dental analogy: you do floss refactoring in order to avoid having to do root canal refactoring. The refactoring patterns suggested in this report are more toward the more invasive kind.

The more dominant works currently in play in regards to concrete patterns and techniques are Feather's [8] and Fowler's [9]. Feather's is more closely related to the topic of this report (that of legacy code) while Fowler's is more toward general refactoring.

Fowler offer some 80 refactorings spanning from "*Add Parameter*" [9](p. 275) to "*Tease Apart Inheritance*" [9](p. 362) and is hence a great source for general patterns. The collection offers a comprehensive guide on how to use refactoring to clean up code, making the intent of the code more explicit and improving readability, maintainability and code structure.

Feather's publication focuses more on how to introduce new features, create seams for decoupling and bring code under test. Even though comprehensive, it offers a more specialized set of patterns.

This report deliberately avoids recapping the patterns in these two works and the reader is encouraged to seek them out as further reading.

3.4.1. Internal Refactoring

Internal refactoring refers to the actions covered by the more strict definition provided by Fowler, *3.4 Refactoring*. Hence, it refers to changing the internal structure of a class, or function without changing its public signature or apparent behavior.

3.4.2. External Refactoring

External refactoring refers to the actions that come into play by the relaxation introduced by the definition used in this report. That includes, the extraction of interfaces, encapsulation of member variables, etc; any changes that syntactically affect other classes/components but do not change the intended behavior. This is done without tests in place since the assumption is that this type of refactoring is done with the explicit purpose of getting code to a testable state.

3.5. Unit Testing

A unit test is a simple piece of code that tests a small unit of code such as a method, a set of methods that together services a specific task. A class or method that is exercised by unit tests is said to be under test. Bringing a class/method under test is the action of writing such a test and setting it up so that the class/method is tested satisfactory.

code example I. Unit Test

```
void MathLibTest::TestAbsWithIntegers()
{
   CPPUNIT_ASSERT_EQUAL(2, MathLib::Abs(2));
   CPPUNIT_ASSERT_EQUAL(0, MathLib::Abs(0));
   CPPUNIT_ASSERT_EQUAL(2, MathLib::Abs(-2));
}
```

An example of how a math lib test could look like using CppUnit; testing the absolute function.

Traditionally, unit tests where written by and performed by testers, not the developers who actually wrote the code. That has changed with the test first paradigm introduced by Test Driven Development. Unit testing in the context of TDD is done by the developer, in conjunction with writing the actual code –see *3.2 Test Driven Development*.

Usually a framework is used to help with the testing. The most common ones follow the same pattern and are collectively known as xUnit. The perhaps most widely used for java is JUnit, for C# NUnit, for C++ CppUnit, among others.

The fundamental principle behind the xUnit test pattern is to verify that the class or function under test are in a specific state, has produced a specific result or has performed a specific task. This is done using what is called Asserts[7]. As in *code example I* the Assert verify a specific condition such as that the actual result equals the expected result. [2] (p. 157)

3.5.1. Properties of Unit Tests

According to Meszaros in [25], unit tests should be fully automated and do a round-trip test against the class or component being tested through its public interface. By ensuring that each test is a single-condition test that exercises a single method or object in a single scenario, defect localization can be achieved. They should also be isolated and independent from each other. They should be able to be run in any order with the same result.

[7] Asserts in unit testing frameworks are not to be confused with the traditional c assert. Asserts used for testing do not halt the program. It merely reports the failure to the test runner and aborts the given test. It does not halt the entire test run. Since tests are written to be independent, the next test in the same test suite can safely be run.

Unit tests should further be self-checking. The expected outcome should be expressed in the test so verification becomes part of the test. The test should only report pass or fail. The developer should not be required to verify the output of the interaction taking place within the test.

Another aspect is speed. Unit tests should be fast enough to be run often. Preferably every time you compile. They should not take more than 1-2 seconds in total, absolutely not more than 10. Hence, individual tests should run on a few milliseconds.

According to some sources [28], [56], Unit testing as part of TDD is slightly different from traditional unit testing in a strict test as verification sense. Since TDD focus on design rather than verification it also reflect on how tests are written. TDD tests tend to expand *unit* to refer to a segment of code that has a single purpose, and can hence span over multiple classes. In strict unit testing, all dependencies should be replaced with test fakes so a class can be tested in complete isolation.

3.5.2. Test Doubles, Fakes, Stubs and Mocks

Using the definition of Roy Osherove, chief architect at Typemock, *Fakes* is the common term for *Stubs* and *Mocks*. *Stubs* refer to an object that acts as a stand-in for a real object in an isolated test scenario. The stub object can serve a limited set of expected data required in the test scenario. In contrast, in addition to being a stand-in for a real object, *Mocks* also record operations that are done to the object so that they can be asserted against that they actually took place during the test.[34]

Slightly different but mainly complementary terminology such as test doubles, dummy object, test spy, among others, is used in [25] to further categorize these kinds of objects. However, these detailed distinctions are not made here. For the context of this report, Osherove's definition is used with the addition of *Test Double* as a synonym for *Fake*.

In TDD unit testing, fakes are only used as a way of keeping the tests fast and to eliminate complexity. Hence, uses of any resource not in memory (here referred to as an expensive resource) such as files, database access, networking, among others are usually replaced with fakes.

Roy Osherove:

"*… as far as I'm concerned as long as all the code under test is under your control, it's your code and it runs in memory and is repeatable and doesn't touch external states such as databases, file systems or even statics, it could be considered a unit test*" [34] (time: 09:30)

3.5.3. Friendlies

Misko Hevery introduced the term *Friendly* to denote an object used in conjunction with TDD testing [41]. A friendly is a fake or a real object that has been configured so that it is suitable for testing. This could for example be a cache object that in the real application is configured to handle thousand of entries while in the testing scenario only holds a few. With only a few entries it is fairly easy to instrument a cache hit and cache miss, which in turn can be used to exercise the class under test.

A problem with many classes is that they are not written with being friendly in mind. Often are `#define` statements are constants used to configure classes. That however, does not allow for friendly configuration in a test case without resorting to changing code. With a little care, though, it is quite easy to do. In the example below, a traditional version and a friendly version of a cache class are outlined. Notice how the simple use of templates makes the class just as type safe, but configurable.

code example II. Friendly Classes

```
#define NR_OF_CACHE_ENTRIES 1000
class TraditionalCache
{
public:
   void* getChachedObject(int id);
private:
   void* m_cacheList[NR_OF_CACHE_ENTRIES];
};
```

```
template<unsigned long _cache_size>
class FriendlyCache
{
public:
   void* getChachedObject(int id);
private:
   void* m_cacheList[_cache_size];
};
```

```
{
   // production case
   TraditionalCache tc;
   FriendlyCache<NR_OF_CACHE_ENTRIES> fc;

   // test case
   // TraditionalCache tc ??
   FriendlyCache<2> cache;
}
```

3.6. Dependency/Coupling

Throughout this report, the concept of dependency is often referenced. To make the discussion meaningful, the terminology has to be clearly mapped out. Dependency or coupling can be defined as "*the degree to which each program module relies on each one of the other modules*" [70][8]. This definition can be expanded to a more granular level to encompass components, classes and functions.

In this context, **X** *uses/depends upon* ***Y*** can refer to (but not limited to):

- Inheritance: **X** inherits from **Y; X** is a subclass of **Y**.
- Member variable: **X** has a member variable of type **Y**.
- Type use: **X** is a function or class with method that uses local variable of type **Y**.
- Function call: **X** is calling function **Y**.
- Parameter in signature (function): **X** is a function with parameter of type **Y**.
- Parameter in signature (class): **X** is class with method that has parameter of type **Y**.
- Global instance: **X** uses global variable of type **Y**.

Coupling as a software quality metric was invented by Larry Constantine, one of the originators of Structured Design, where *low coupling* (also *loose* and *weak*) is often a sign of good design [70]. The exact definition and formula for calculating coupling exists in many variations. One model suggested in [78] according to [70] is presented below.

For data and control flow coupling:

di: number of input data parameters

ci: number of input control parameters

do: number of output data parameters

co: number of output control parameters

For global coupling:

[8] [70] attributes coupling and cohesion to Larry Constantine, [79]

gd: number of global variables used as data

gc: number of global variables used as control

For environmental coupling:

w: number of modules called (fan-out)

r: number of modules calling the module under consideration (fan-in)

A variable can be used for both data and control. Data refers to values used as part of a computation, while control parameters are used in condition structures such as decide the outcome of an `if` statement.

$$\text{Coupling (C)} = 1 - \frac{1}{d_i + 2c_i + d_o + 2c_o + g_d + 2g_c + w + r}$$

However, the coupling metric has evolved and is now combined with other metrics such as cohesion to signify object oriented design quality as not all dependencies are bad. All components, have to have some kind of coupling to other code segments, otherwise they serve no purpose and can be removed. The combination of metrics to validate good OOD is still debated upon. The set of metrics used in this project are described in *2.4 Quality Metrics*.

The first distinction made is in the direction of the coupling itself:

Afferent coupling (C_a): inbound coupling, the use of the component/class/function elsewhere.

Efferent coupling (C_e): outbound coupling, the use of other components/classes/functions by the component/class/function itself.

It can be thought of as arrows pointing toward the depended upon items. Afferent coupling is the number of arrows pointing inward to the item itself. Efferent coupling is the number of arrows pointing outward toward other components/classes/functions.

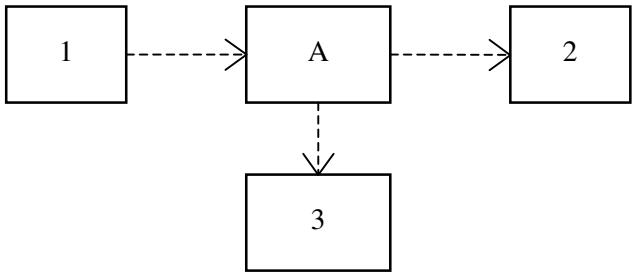


figure V. Afferent and Efferent Coupling

In *figure V*, **1** uses **A** and hence depends on **A**. **2** and **3** are used by **A**. The number of afferent couplings for **A** is one because of **1**. **A**'s efferent couplings are two, since **A** uses **2** and **3**.

For **A**: $C_a = 2 \quad C_e = 1$

Code segments that have neither afferent nor efferent couplings are likely dead and can often be deleted.

Martin applies afferent and efferent coupling in the context of Booch Class Categories [73] but it can easily be expanded to adhere to different levels of granularity and can refer to coupling among, assemblies, components, classes, methods/functions, types, etc.

Martin uses afferent and efferent coupling to define a metric describing instability:

I: Instability: $\frac{C_e}{C_a + C_e}$

This gives a range of [0,1] where I=0 indicates maximal stability and I=1 indicates a maximally instable category [18] (p. 6).

The second aspect of coupling is what is depended upon. Here we use a variation of Martin's classification of Abstractness:

$$A: \textit{Abstractness:} \frac{\#\,abstract\ classes\ in\ category}{total\ \#\,classes\ in\ category}$$

This gives a range of [0, 1] where A=0 means concrete and A=1 completely abstract. [18] (p. 6)

As in the case of Instability, Martin applies Abstractness on Booch Class Categories [73], to describe dependencies between categories.

Here, the concept is expanded to concern individual classes. Instead of classes in categories, the measurement can be applied to abstract methods in a class. Hence, a class with only pure virtual methods is completely abstract; both in terms of the metric and the semantic meaning.

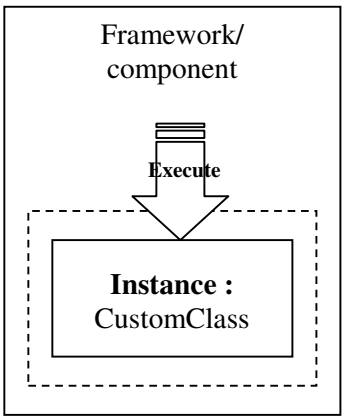
Martin concludes that categories should preferably be either maximally stable and abstract or maximally instable and concrete [18] (p. 6). Translated to the class level this yields that code that change should be concrete and in turn depend on abstractions that are stable. These kinds of dependencies should be preferred.

Loosely coupled code has two attributes of importance. It has few couplings and the couplings that exists are of the preferred nature –instable concrete items depends on stable abstract items [67]. Hence when talking about breaking dependencies it refers to the act of improving those attributes; refactoring code so it becomes more functional cohesive and thereby reducing the sheer number of couplings and replacing the remaining ones to depend upon abstractions. Alternatively, other mechanisms can be used to achieve the same result as depending upon abstractions.

3.7. Inversion of Control (IoC)

According to Martin Fowler [60] Inversion of Control originates from the "Hollywood Principle"[9] probably introduced by Richard E. Sweet in [81]. The term Inversion of Control may have first been introduced by Johnson and Foote in [76]. The concept stipulates an inversion of who controls the program flow and calls the actual methods/functions. Originally, it was most common in frameworks where the developer would construct a class or function, often implementing a given interface and inject the framework with an instance of that class. The framework would then call methods on the instance as needed.

figure VI. General IoC



In later years, with the rising popularity of the Spring Framework for Java and the emerging of other IoC frameworks and containers, the meaning of Inversion of Control has somewhat changed. It has been extended to also denote the inversion of control of creation, configuration and object (lifecycle) lifetime management. This is the definition assumed throughout this report.

Ke Jin, maintainer of PocoCapsule, describes it as "*injection/setting scenario is an **inverse** version of the traditional lookup/resolving scenario*". [53]

The implication of this statement is that a client object, instead of **pulling** in depended upon objects, gets the dependencies **pushed** onto it. The control of the action of creating and binding the dependencies are inverted from the client object to an external entity.

[9] "Hollywood Principle: "*Don't call us, we'll call you*"

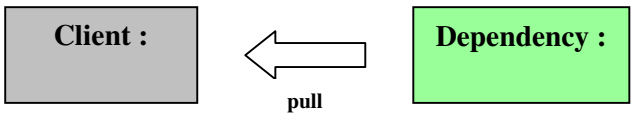


figure VII. Client Object Pulling in Dependencies

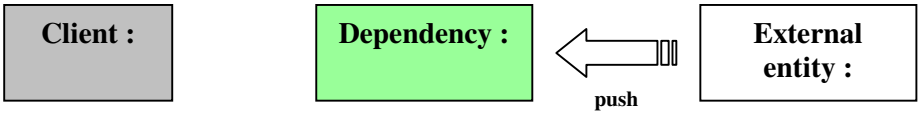


figure VIII. IoC: Pushing Dependencies onto Client Object

A way of facilitate this is a design pattern called Dependency Injection (DI). Instead of letting object create or get their dependencies/resources themselves, as in accordance with the principle of RAII (see *3.12 RAII – Resource Acquisition Is Initialization*) dependences are injected using various methods discussed in *3.8 Dependency Injection (DI).*

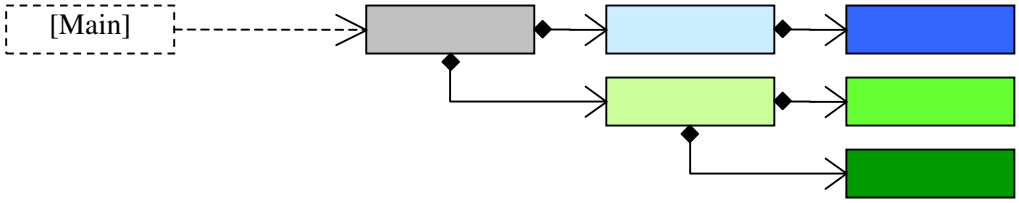


figure IX. Traditional Object Ownership Graph

This result in a change of the ownership graph from the one in *figure IX* to the one in *figure X*.

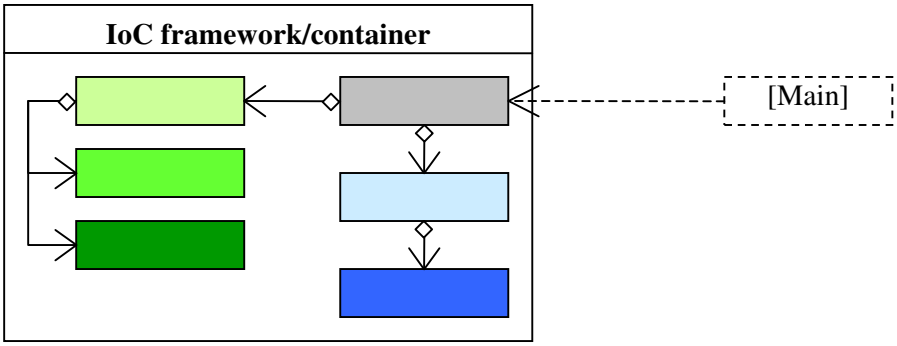


figure X. Object Ownership Graph with Inversion of Control

A common confusion is that IoC in itself is a design pattern, that it is equivalent to DI. This is however not the case. IoC is a high level principle which primarily serves the purpose of isolating components. The use of DI as means of wiring components together should not be mistaken to hold any value in itself; it is merely a consequence of the isolation. Stefano Mazzocchi, former Director of Apache Software Foundation, puts it this way:

"IoC is about enforcing isolation, not about injecting dependencies. The need to inject dependencies is an effect of the need to increase isolation in order to improve reuse, it is not a primary cause of the pattern"[26] (p. 7)

The point of isolation is that it gives loosely coupled code and hence makes it easier to test. This, since any depended upon component can easily be replaced with a fake object for testing purposes as the control of creation lies outside the component. See also section *3.3 Testability*.

Further, by moving the control of creation away from the components themselves, general logic and object graph creation can be cleanly separated. This in turn, allows for easy configuration and replacement of components, making the application easy to extend without modifying existing code. [24](p 154-156)

In contrast to other facilitating patterns such as *Service locator*[10], IoC is non-invasive in the sense that it does not create dependencies to the facilitator, require that components themselves inherit from a specific interface or are otherwise decorated.

Contrary to popular belief, IoC's innovation is not to reduce component-component coupling. Service locator already solved that for us. The change that came about with IoC is that it eliminates component-container coupling.

figure XI. Component-Component Dependency

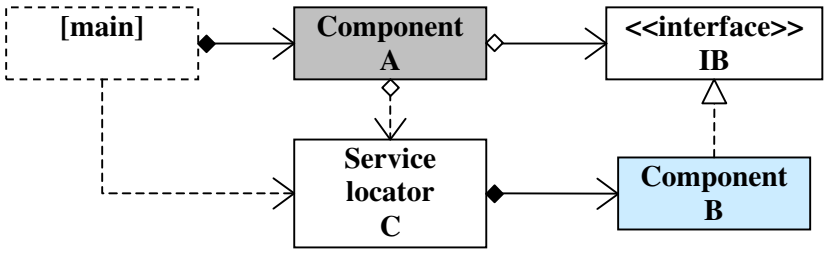


figure XII. Service Locator Dependency

An example: Component **A** depends on **B**. To decouple **A** from **B**, in the case of the service locator, **A** now asks service container **C** for **B**. Hence **A** depends on **C**. In the case of IoC, **A** doesn't depend on the facilitating IoC container, nor **B**. **A** and **B** are completely isolated from each other as well as the container. This since **A** is given **B** by the container **C** without having to know about **C**.

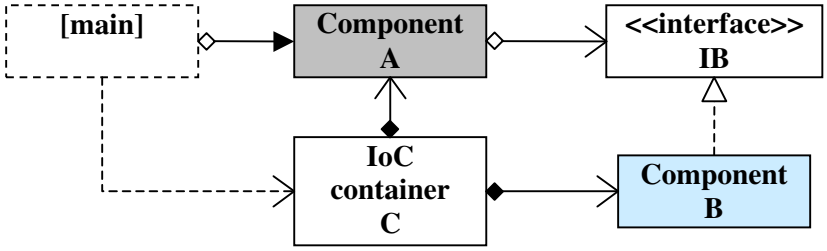


figure XIII. IoC Container Dependency

In the above example, the concept of IoC container is brought forth. It is further explored in section *3.9 Inversion of Control Containers*.

Components abiding by the IoC principle should hence only declare their needed dependencies and configurations. Not get them themselves. This will result in more reusable classes, classes that are easier to test and systems that are easier to assemble and configure.[51]

[10] Service Locator is a design pattern used to make services/components available to other components. The abstraction is used so eliminate the strong coupling between the components. A description can be found at [38]

3.8. Dependency Injection (DI)

Dependency Injection (DI) is a design pattern where an object is given its dependencies from an outside controlling entity rather than creating or requesting them itself –the dependency is injected. The term is attributed to Martin Fowler and can be found in [38].

The pattern came about as a clarification on what was thought of at the time as a subset of IoC. The name accurately describes a common way to wire components together. Originally, three types of injection were identified. They were unfortunately named simply IoC 1, 2 and 3. The terminology of IoC comes from that IoC containers such as Spring and PicoContainer where using these methods to wire components together. With Fowlers article, injection types were given proper names.

The different ways of injecting components are here collectively denoted *injection paths*. This is a terminology not used elsewhere but created for the context of this report.

In section *3.7 Inversion of Control (IoC)*, it was stated that the use of DI as a means of wiring components together does not have any value in itself. That DI was merely a means to an end. DI does however have value in other contexts. One of these values is that DI makes dependencies explicit. All dependencies are immediately made visible by just looking at the constructor, or the class' methods.

In the following subsections describe the various types of injection. The examples are intentionally kept simple. The purpose is merely to describe the mechanisms. A more thorough discussion is provided in section *7.1.4 Dependency injection: Preferred Methods*.

3.8.1. Constructor Injection

Using constructor injection, any dependencies are injected as arguments to the constructor. In the example that follows, the class `PersistentPersonnelManagerWithAudit` uses a database and a logger. `PersistentPersonnelManagerWithAudit` holds pointers to interfaces that the classes `SQLDatabase` and `AuditLogger` implements. As seen, instead of `PersistentPersonnelManagerWith-Audit` creating its own dependencies, they are handed to it as arguments to the constructor. In a test scenario, it is trivial to use a *fake* or *friendly* implementation of database and logger instead of using the production classes. Logger writing to file or a data retrieved by accessing a database over network will be significant slower and require more setup. Further, it is also quite easy to replace the SQL-database implementation with a more lightweight file based version if it would turn out to be appropriate in a new production setting.

code example III. Constructor Injection

```
class PersistentPersonnelManagerWithAudit
{
public:
   PersistentPersonnelManagerWithAudit (IDatabase* db, ILogger* log)
      : m_db(0), m_log(0)
   {
      m_db = db;
      m_log = log;
   }
private:
   IDatabase* m_db;
   ILogger* m_log;
};
```

```
{
   IDatabase* db = new SQLDatabase();
   ILogger* log = new AuditLogger();
   PersistentPersonnelManagerWithAudit* ppmwa
      = new PersistentPersonnelManagerWithAudit(db, log);
   ... //do some work
   delete ppmwa;
   delete log;
   delete db;
}
```

3.8.2. Setter Injection

Setter injection relies on the use of so called setter methods. It is simply a method that is used to give a value, reference or pointer to a member variable. In the example that follows, the class `PersistentPersonnelManagerWithAudit` on a database and a logger. `Persistent-PersonnelManagerWithAudit` holds pointers to interfaces that the classes `Database` and `Logger` implements. As seen, instead of `PersistentPersonnelManagerWithAudit` creating its own dependencies, they are passed to it using the set-methods.

code example IV. Setter Injection

```
class PersistentPersonnelManagerWithAudit
{
public:
   PersistentPersonnelManagerWithAudit () : m_db(0), m_log(0){}
   setDatabase(IDatabase* db) {m_db = db;}
   setLogger(ILogger* log) {m_log = log;}
private:
   IDatabase* m_db;
   ILogger* m_log;
};
```

```
{
   IDatabase* db = new SQLDatabase();
   ILogger* log = new AuditLogger();
   PersistentPersonnelManagerWithAudit* ppmwa
      = new PersistentPersonnelManagerWithAudit();
   ppmwa->setDatabase(db);
   ppmwa->setLogger(log);
   ... //do some work
   delete ppmwa;
   delete log;
   delete db;
}
```

3.8.3. Interface Injection

Interface injection uses, as the name suggests, an interface as the injection mechanism. An interface is defined that allows for the injection of the depended upon component. The dependent component implements said interface. Interface Injection is very similar to setter injection in terms of the lingual mechanism, but thanks to the interface, a framework or other entity that knows about the interface can handle the injection.

code example V. Interface Injection

```
class PersistenceAspect
{
public:
   virtual void injectDatabase(IDatabase* db) = 0;
};
```

```
class LoggingAspect
{
public:
   virtual void injectLogger(ILogger* log) = 0;
};
```

```
class PersistentPersonnelManagerWithAudit
   : public PersistanceAspect, public LoggingAspect
{
public:
   void injectDatabase(IDatabase* db){m_db = db;}
   void injectLogger(ILogger* log){m_log = log;}
private:
   IDatabase* m_db;
   ILogger* m_log;
};
```

```
class AspectInjectionFactory
{
public:
   AspectInjectionFactory()
   {
      m_db = new SQLDatabase();
      m_log = new AuditLogger();
   }
   ~InjectionFactory()
   {
      delete m_db;
      delete m_log;
   }
   void injectPersistenceAspect(PersistenceAspect* pa)
   {
      pa->injectDatabase(m_db);
   }
   void injectLoggingAspect(LoggingAspect* la)
   {
      la->injectLogger(m_log);
   }
private:
   IDatabase* m_db;
   ILogger* m_log;
};
```

```
{
   PersistentPersonnelManagerWithAudit* ppmwa
      = new PersistentPersonnelManagerWithAudit();
   AspectInjectionFactory f;
   f.injectPersistenceAspect(ppmwa);
   f.injectLoggingAspect(ppmwa);
   ... //do some work
   delete ppmwa;
}
```

3.8.4. Template Injection

Template injection (TI) is not established terminology but introduced here in this report for the use of the C++ template engine to inject dependencies. The almost non-existent interest in template injection could be the lack of general availability or for historical reasons. Java, where DI is most prominent, did not have generics, the closest to an equivalent, until J2SE 5.0. Another reason could be the imposed limitations that templates bring. See section *7.1.4 Dependency injection: Preferred Methods* for further discussion on the ramifications of Template injection.

Template injection makes use of the C++ template engine as a facilitator for the injection of interchangeable component types. Below follows a brief code example showing its use.

code example VI. Template Injection

```
template<class DB, class LOG>
class PersistentPersonnelManagerWithAudit
{
public:
   PersistentPersonnelManagerWithAudit()
private:
   DB m_db;
   LOG m_log;
};
```

```
PersistentPersonnelManagerWithAudit<SQLDatabase, AuditLogger> ppmwa;
... //do some work
```

As seen, with template injection, the extra layer of interfaces for the depended upon components can be neglected. However, the components cannot be replaced at runtime.

3.8.5. Other Types of Injection

Not only data and objects can be injected. Function/Method-injection can be used using the mechanisms of construction injection or setter injection. Pointers to functions and methods on classes can be injected. In C# you have delegates and anonymous functions that can serve the same purpose.

There are additional variations of the previously explained types of DI, some of which are explained at PicoContainer's website [96].

Dynamic languages such as Python and Ruby can modify objects' signature[11] in runtime and hence don't need explicit injection paths such as adapted constructors or setter methods. Other message-dispatched based object models, such as the one suggested in the Wizard book [82], do also allow for runtime modifications without the need of explicit injection paths.

[11] An object's signature is decided by its methods' signatures, what interfaces it implements, how it syntactically looks to the rest of the world. A method's signature, or rather function signature as it is actually called, is the function's extended name. That is the function name along with its parameter types –excluding the return type.

Languages supporting macros such as C, C++ and Scheme can utilize them as an implicit injection mechanism by redefining declarations.

Implicit injection mechanisms all negate one of DI's most important features, namely that classes have to be open about their dependencies. Implicit injection is considered outside the scope of this report.

3.9. Inversion of Control Containers

An IoC container can be summarized as a builder component or framework that realizes the inversion of control concept. Its purpose can be recapped as four responsibilities (sometimes listed as three):

- create components
- configure components
- (wire components together)
- manage component life time

To be considered a true IoC container it should be non-invasive. Ke Jin, maintainer of PocoCapsule, describes it as:

"*By being non-invasive, decent IoC frameworks are neutral to component models. They impose neither any mandatory interfaces nor operation signatures on components nor framework specific home interfaces. Rather, IoC frameworks accept almost all types of programming artifacts as components and plain old instantiation/lifecycle control methods (such as constructors/destructors, factories, reference counting methods, etc.).*"[53]

IoC containers, in the form they exist today, first emerged from the Java community in the form of the Spring framework and PicoContainer. Later Hivemind (now retired) and Guice would follow. In the .NET world we have Castle Windsor, StructureMap, Spring.NET, Unity, PicoContainer.NET among others [92].

A common trait among many of the modern IoC containers mentioned above is that they rely on reflection. The lack of comparable language feature in C++ could help explain the relatively meager list of IoC containers for C++. Two of the available frameworks will be evaluated later in this report.

3.10. Injectables and Newables

In [42] Misko Levery, introduced the terms *injectables* and *newables*. Injectables refer to a class of types that should be injected –see section *3.8 Dependency Injection (DI)*, while newables are types that cannot be supplied by DI frameworks or IoC containers. Injectables tend to have interfaces for easy replacement in testing scenarios.

Levery gives the example of `AudioDevice` as an injectable to `MusicPlayer`, while creating a new email-address based on a user provided string is a Newable [42].

Once the distinction is made, a set of guidelines for the two classes of types can be supplied. Levery suggests that injectables can be injected with other injectables, but not with newables. The reason for this is that DI frameworks and IoC containers do not know how to supply newables. Similarly, newables can depend on and obtain ownership over other newables on construction but not over injectables. Newables in general has a different lifetime than injectables and therefore should not *own* them. Newables may know about injectables but never own them and be responsible of disposing them.

To keep the abstractions, newables can be wrapped in factories so injectables can ask for them during the normal run of the application –the factory itself being an injectable.

To illustrate this principle, an example from Codebase II is used. Codebase II is a simple 2D top-down scrolling game. Among other components, it consists of a `GameObjectFactory` that creates and destroys game objects. There is also a `GameObjectManager` that handles all game objects in play. `GameObjectFactory` and `GameObjectManager` are typical injectables. They are part of the overall program architecture and known at initialization.

During gameplay, the user can do certain actions that trigger the creation of new game objects. For instance, the player object consists of a hovercraft with a mounted gun to shoot down pirate ships and rouge military vessels. The game has no way of knowing in advance how many, when or where shots will be fired. Hence, they need to be created at runtime – a typical Newable. Instead of creating the shot object itself, `GameObjectManager` will ask the factory to supply it.

code example VII. Injectables and Newables

```
{
   //Initiliazation, can be performed by IoC containers. Hence Injectables
   GameObjectFactory* gof = new GameObjectFactory();
   GameObjectManager* gom = new GameObjectManager(gof);
}
```

```
/*
Inside GameObjectManager
m_gof is a member variable pointing to a GameObjectFactory
GameObjectManager has an internal cache of player-shots.
If all cached shots already are in use, the GameObjectManger tells the
factory to create a new shot.
*/
Shot* GameObjectManager::AddPlayerShot()
{
   if (m_playerShotsInUse == m_playerShots.size())
      m_playerShots.push_back(m_gof->getPlayerShot());

   // 0-index, hence increment after read of value
   Return m_playerShots[m_playerShotsInUse++];
}
```

3.11. Object Oriented Design and Its Effect on Testability

The heart of Object Oriented Design (OOD), its core purpose, is managing dependencies [61](time: 00:02:30). The claims in previous sections that test first programming encouraging good object oriented design comes from that the practice makes it easier to manage dependencies. It promotes a program structure of loosely coupled, cohesive components which in turn results in high testability.

In this section, the attributes and explicit mechanisms of such a design is explained. This dissection is needed for the discussion featured in *7 Results*.

In [61] Robert C. Martin claims that poor dependency management results in rigid, fragile code that cannot be reused and is hard to test. By rigid, Martin means that the code is hard to change and that the impact of a change cannot accurately be predicted. If not predicted, the job cannot be estimated and as consequence time and cost cannot be quantified. Hence, managers become reluctant to authorise change and therefore, the code does not allow itself to change.

Using the terminology of *Instability* and *Abstractness* established in *3.6 Dependency/Coupling*, highly stable and concrete components are rigid.

"*It cannot be extended because it is not abstract. And it is very difficult to change because of its stability.*" Martin - [18] (p. 7)

The fragile attribute, according to Martin [61] , results in that a single change requires a cascade of subsequent changes. New errors appear in areas that seem unconnected to the changed areas, quality becomes unpredictable and thus the development team loses credibility.

Strongly coupled code makes it difficult to take desirable parts of a design and leave undesirable parts. The work and risk of extracting the desirable part may exceed the cost of redeveloping from scratch, and hence reuse becomes very limited. [61]

In the following subsections, five design principles are presented. Robert C. Martin has written papers on these principles in *The C++ Report* as well in several of his books. The principles are briefly explained here as they relate to the results presented later in chapter 7. In late they have been dubbed the SOLID principles, but are here presented in their original order.

3.11.1. Single Responsibility Principle

Martin attribute the Single Responsibility Principle (SRP) to the work of Tom Demarco [75] and Meilir Page-Jones [77], there presented in terms of cohesion [23] (p. 95). For the principle itself, Martin summarize it as "*A class should have only one reason to change.*" [23] (p. 95)

By that, he means that if a class has more than one reason to change, it usually means that the class has more than one responsibility. These responsibilities may often come from different aspects or parts of the application. This means that a change brought about from one part of the program risk affecting the other part due to the coupling through the common class. This produces a rigid and fragile system. [23] (p. 95-98)

3.11.2. Open Closed Principle

Bertrand Meyer:

"*Modules should be open for extension, but closed for modification*" [83]

Martin credit Meyer with the Open Closed Principle, OCP [23] (p. 99). The principle states that modules should be open for extension in the form of allowing for change of its behavior. They should also be closed for modification in the sense that neither existing source or binary code should need to change. Hence, adding new features or changing behavior comes by adding new code, not editing existing code. [23] (p. 99-109)

This is usually accomplished with the use of abstractions through interfaces or abstract base classes and inheritance. Polymorphism then allows for the introduction of new types of objects without the need to change the original code.

The example immediately below shows a code snippet that does not abide by this principle. Here we have a common `Control` with two subtypes: `TextBox` and `Button`. The controls can be drawn using corresponding functions. There is also a `DrawAllControls` function that accept a list of controls.

code example VIII. Open Closed Princle

```
typedef pair<int, int> Point;
enum ControlType {TextBox, Button};
typedef struct Control* ControlPointer;
```

```
struct Control
{
   ControlType type;
};
```

```
struct TextBox
{
   ControlType type;
   string text;
   Point position;
};
```

```
struct Button
{
   ControlType type;
   string caption;
   Point position;
   bool clicked;
};
```

```
void DrawTextBox(struct TextBox* textbox); //defined elsewhere
void DrawButton(struct Button* button); //defined elsewhere
```

```
void DrawAllControls(ControlPointer list[], int n)
{
   for (int i = 0; i < n; i++)
   {
      ControlPointer control = list[i];
      switch(control->type)
      {
      case TextBox:
         DrawTextBox((struct TextBox*) control);
         break;

      case Button:
         DrawButton((struct Button*) control);
         break;
      }
   }
}
```

Besides adding corresponding draw function, every time a new control type is added the `enum ControlType` has to be updated. The `DrawAllControl` function also has to change. Obviously this violates the principle. Otherwise `drawAllControls` would not have to change.

The next example uses polymorphism and interface inheritance to solve the same problem and now abiding by the principle. In this new version new types of controls can be added without requiring modifications to `drawAllControls`.

code example IX. Open Closed Principle (cont.)

```
class Control
{
public:
   virtual void Draw() const = 0;
};
```

```
class TextBox : public Control
{
public:
   virtual void Draw() const;
};
```

```
class Button : public Control
{
public:
   virtual void Draw() const;
};
```

```
void DrawAllControls(list<Control*>& l)
{
   for(list<Control*>::iterator it = l.begin(); it != l.end(); it++)
   {
      (*it)->Draw();
   }
}
```

The principle can also be extended to explain some of the common heuristics of OOD, such as keeping member variables private (encapsulation), avoiding global state and not depending on runtime type information. Without abstractions changes leak onto anything that depend upon them. [22] (p. 9-12)

3.11.3. Liskov Substitution Principle

Liskov Substitution Principle (LSP), invented by Barbara Liskov in 1988 and regards to inheritance, how subclasses should relate to their base classes.

Martin explains it as: "*Subtypes must be substitutable for their base types*" [23] (p. 111)

Substitutability can be divided into syntactical and semantic substitutability. Syntactical substitutability can be formalized as a set of requirements on the new type. A type is considered to abide by the syntactic contract and hence is substitutable for another type if the new type's:

- input types are contravariant
- output types and exceptions are covariant

to the original type. In this context contravariant means that the type is the same or a superclass to the original type and covariant is that the type is the same or a subclass to the original. [27]

Semantic substitutability, sometimes referred to as *strong behavioral subtyping* [71], is more difficult to formalize. It holds several nuances and the example that follows below will make an attempt to highlight some of them.

Martin's presentation of LSP holds a strong focus on substitution in the meaning of following expected behavior. For further reading, [19] and [23] (p. 111-125) are suggested.

However, the following example, a variation on one of Martin's examples in [19], shows one of the aspects of LSP. In this case it shows how a subtype breaks the expected behavior of that type.

In the example there is a rectangle class where the width and height can be set independently. There is also a function that is part of a transformation library that can scale rectangles. The function will simply scale the width and height in accordance with the provided factor. A test showing that it works is also provided.

code example X. Liskov Substitution Principle

```
class Rectangle
{
public:
   Rectangle(Point topLeft, double width, double height)
      : m_topLeft(topLeft),
        m_width(width),
        m_height(height)
   {}
   void setWidth(double width) {m_width = width;}
   void setHeight(double height) {m_height = height;}
   double getWidth() const {return m_width;}
   double getHeight() const {return m_height;}
private:
   Point m_topLeft;
   double m_width;
   double m_height;
};
```

```
void ScaleRectangle(Rectangle& rect, double factor)
{
   rect.setWidth(rect.getWidth() * factor);
   rect.setHeight(rect.getHeight() * factor);
}
```

```
void TestScaleRectangleWidthRectangle()
{
   Rectangle rect(Point(1,1), 2.0, 3.0);
   ScaleRectangle(rect, 2.0);
   assert(rect.getWidth() == 4.0);
   assert(rect.getHeight() == 6.0);
}
```

Assume now that a new type is introduced –`Square`. Using the "***is a***" principle for object oriented analysis it can be concluded that since a square ***is a*** rectangle, `Square` should be a subclass of `Rectangle`. The fact that a `Square` only need one variable to store the sides length and not two as in the case of the `Rectangle`, should give of a warning flag, but let assume that memory is not a concern and the "***is a***" principle is adhered to.

Since a square's width and height is always equal, the `setWidth()` and `setHeight()` methods need to be overridden. They are made virtual in the `Rectangle` base class to allow this change and overridden in the new subclass `Square`.

code example XI. Liskov Substitution Principle (cont.)

```
class Rectangle
{
public:
   ...
   virtual void setWidth(double width) {m_width = width;}
   virtual void setHeight(double height) {m_height = height;}
   ...
};
```

```
class Square : public Rectangle
{
public:
   Square(Point topLeft, double side)
      : Rectangle(topLeft, side, side)
   {}
   virtual void setWidth(double width)
   {
      Rectangle::setWidth(width);
      Rectangle::setHeight(width);
   }
   virtual void setHeight(double height)
   {
      Rectangle::setHeight(height);
      Rectangle::setWidth(height);
   }
};
```

All is well, or so it seems. Just to make sure a test of `ScaleRectangle` using a `Square` is added to the test suite.

code example XII. Liskov Substitution Principle (cont.)

```
void ScaleRectangle(Rectangle& rect, double factor)
{
   rect.setWidth(rect.getWidth() * factor);
   rect.setHeight(rect.getHeight() * factor);
}
```

```
void TestScaleRectangleWithRectangle()
{
   Rectangle rect(Point(1,1), 2, 3);
   ScaleRectangle(rect, 2);
   assert(rect.getWidth() == 4);
   assert(rect.getHeight() == 6);
}
void TestScaleRectangleWithSquare()
{
   Square square(Point(1,1), 2);
   ScaleRectangle(square, 2);
   assert(rect.getWidth() == 4);
   assert(rect.getHeight() == 4);
}
```

However, the `assert` will fail. The `ScaleRectangle` function will first scale the width with a factor of 2, hence making it 4. But since `Square` has overridden the `setWidth()` method it will automatically adjust the height to 4 as well. When the `ScaleRectangle` then scales the height by 2, the squares sides will become 8.

If `Square` had been a proper subtype of `Rectangle`, `ScaleRectangle` would have worked with any `Rectangle`. However, since `Square` violates LSP, it is obviously not substitutable and therefore should not have been made a subclass of `Rectangle` in the first place.

3.11.4. Dependency Inversion Principle

Martin summarizes the Dependency Inversion Principle (DIP) in two statements:

A. *High level modules should not depend upon low level modules. Both should depend upon abstractions.*

B. *Abstractions should not depend upon details. Details should depend upon abstractions.*

Martin - [20](p. 6)

Martin claims that traditionally structured programs tend to have reusable low level parts (if well designed) but the high level layer tend to depend on the lower levels. This results in a structure where business rules and policy structures (high level modules) depend on low level modules and therefore cannot be reused.

By inverting the dependency so that the high-level modules provides a service interface they need fulfilled by lower level modules, high-level module reuse can be achieved. If the service interface is made into a neutral abstraction it can be reused by other components and thereby work as a model for collaboration between any numbers of high and low level modules where all classes adheres to this service interface.

A simple implementation of this principle is dependency injection as described in section *3.8 Dependency Injection (DI)*

The following example, inspired by [20] (p. 3-6) illustrates its use to create a reusable high level module.

Taking a simple program as an example, one that copies characters entered on the keyboard to the screen. The keyboard reader and screen writer modules are reusable low level modules of the kind often encountered in programs. However, in this first version the high level module, copy is just as rigid as Martin claims, since it depends on the keyboard reader and screen writer modules.

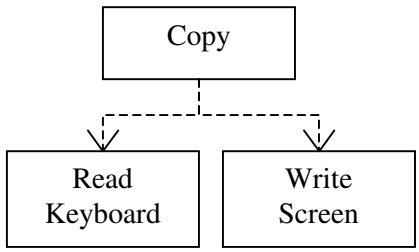


figure XIV. Copy Program: Rigid Structure

code example XIII. Copy Program

```
void Copy()
{
   int c;
   while ((c = ReadKeyboard()) != EOF)
      WriteScreen(c);
}
```

The copy program cannot be used for anything else than reading from the keyboard and write to the screen. If it should be extended to also include output to disk and printer and input from file or network, the copy module need to be redesigned.

However, if neutral abstractions are created, the high level copy module becomes more versatile and reusable.

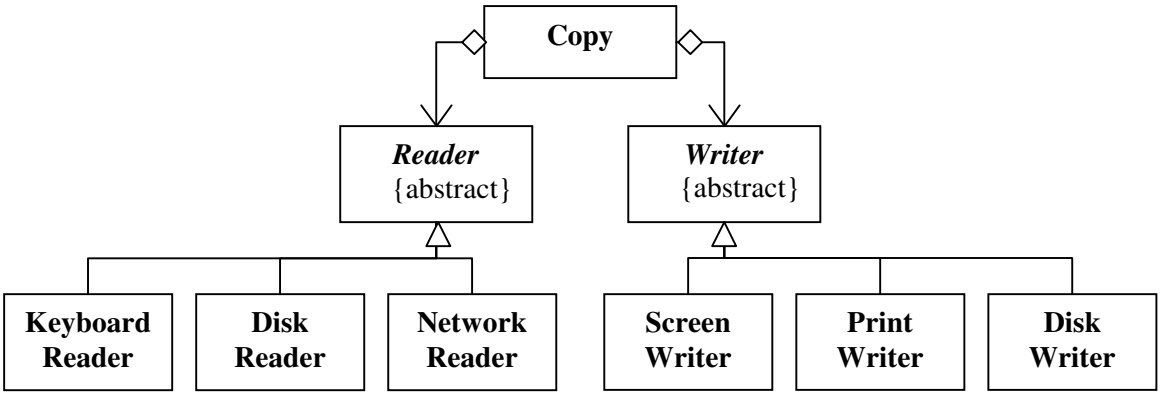


figure XV. Copy Program: Dynamic Structure

Obviously, this is an oversimplified example. It should merely be seen as a statement of principle. The point being that the dependencies are inverted from the copy module depending on low level modules, to the low lever modules conforming to the service interface (Reader, Writer) that the higher level module requests.

3.11.5. Interface Segregation Principle

The Interface Segregation Principle (ISP) is a way of dealing with what Martin refers to as "*fat*" interfaces [23] (p. 135). He further states that:

"*Clients should not be forced to depend upon interfaces that they do not use.*" Martin - [21] (p. 5)

Clients that rely on "fat" interfaces are affected when any single method change in that interface, whether the client uses that method or not. By segregating the interface into smaller parts, each client

is not affected by changes in other clients' way of interacting. Hence the same object displays different interfaces depending on which client it is collaborating with. It is a way of separating concerns.

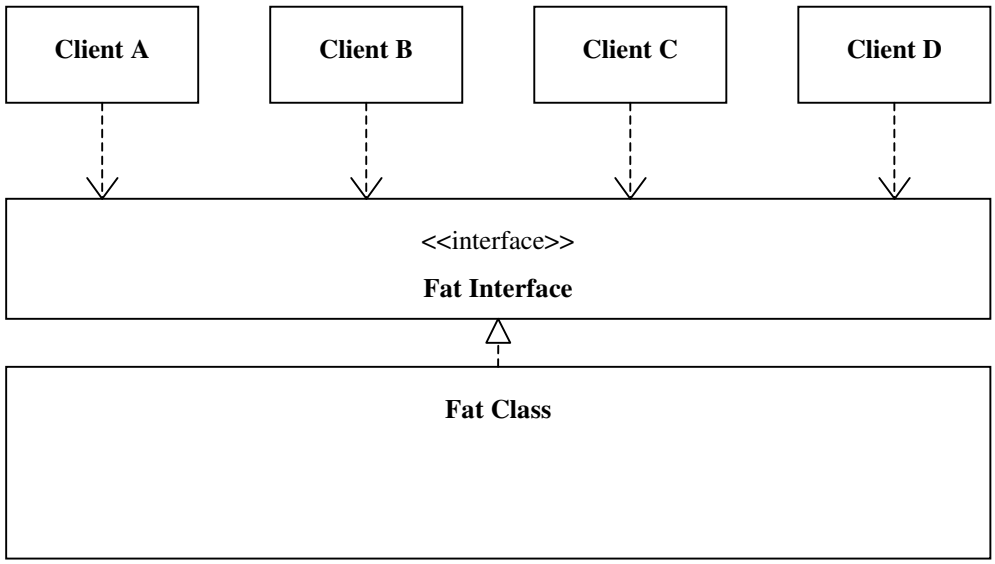


figure XVI. Traditional Fat Interface

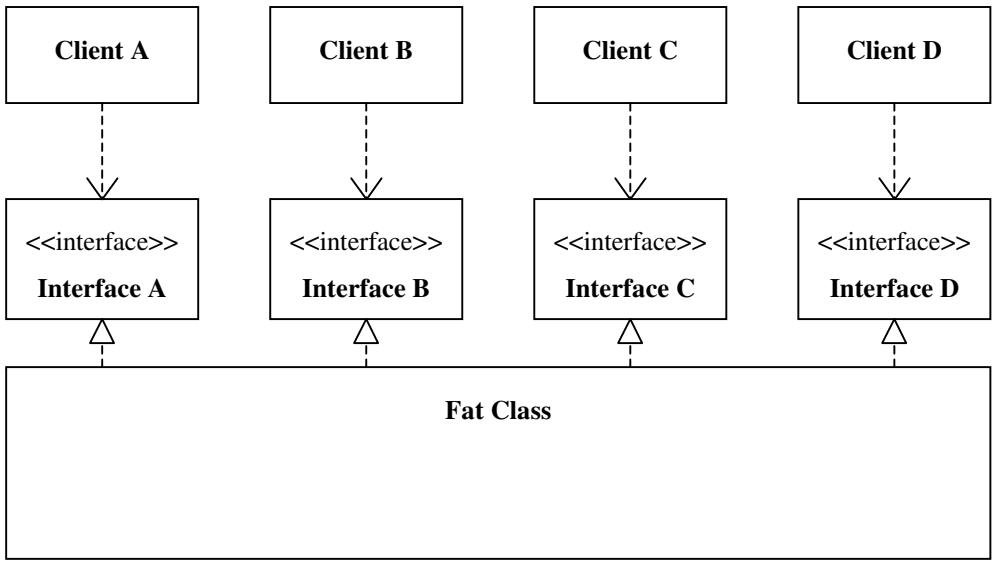


figure XVII. Interface Segregation Principle

3.12. RAII – Resource Acquisition Is Initialization

According to [68], Bjarne Stroustrup claims in [80] to have been the first to articulate the concept of RAII –Resource Acquisition Is Initialization. The point of RAII is that resources should be acquired as part of the initialization process. Further, the resources should then be released in a teardown process.

In C++ this is done with constructors and destructors. Resources are acquired in the constructor and released in the destructor. The strength of this pattern is that if an exception is thrown during initialization, the resource doesn't get acquired. The intended consequence of this in C++ is that the destructor can never be called on an object whose constructor threw an exception. Hence, an object can never be forced to release a resource it doesn't have.

As an example of the power of this pattern, look at the following scenario:

In a multi-threaded environment, a shared resource should only be manipulated while under exclusive control. Otherwise, there is the risk of race condition. The naïve way of doing this would be this:

code example XIV. Acquire Lock

```
{
   lock.acquire();
   try
   {
      ... //do stuff
   }
   catch (exception e)
   {
      lock.release();
      throw; //Error not handled here, hence re-throw
   }
   lock.release();
}
```

This however, is quite verbose and not very well partitioned.

A better way of doing this, following the RAII pattern, would be to encapsulate this in its own class.

code example XV. Acquire Lock – RAII

```
class Resource
{
public:
   Resource(ILock& lock)
   {
      m_lock = &lock;
      m_lock->acquire();
   }
   ~Resource()
   {
      m_lock->release();
   }
   void doStuff()
   {
      ...
   }
private:
   ILock* m_lock;
};
```

```
{
   Resource resource(lock);
   resource.doStuff();
}
```

The intent of the code becomes much clearer.

In the first example, there are numerous ways that release is not going to be called. If the `try-catch` block is omitted and an exception is thrown release would not be called and as a result, the program would most likely end up deadlocking. Further, if the `exception` should be dealt with in the same context, the `release` cannot be temporal coupled to the exception handling (handling must be done before release outside the catch statement) or one has to introduce a mechanism to insure that `lock.release` is not called twice. An early return statement, introduced by a careless programmer,

would also result in that `lock.release` is not called with the possible consequence that the program deadlocks.

In the second example using RAII, if `doStuff()` throws an exception or a `return` statement is added the lock will always get released. As soon as the code block is done executing and the stack is cleared, the destructor of `Resource` will be called and release the lock.

A more common way of seeing the RAII pattern at work is that of objects creating their own dependencies, generating a chain of initialization.

code example XVI. Object Chain Initialization

```
class A
{
public:
   A(string str) : m_b(0), m_c(str)
   {
      m_b = new B();
   }
   ~A()
   {
      delete m_b;
   }
private:
   B* m_b;
   C m_c;
};
```

```
class B{};
```

```
class C
{
public:
   C(string str) : m_s(str) {};
private:
   string m_s;
};
```

As previously concluded in *3.3 Testability* and *3.5 Unit Testing,* this pattern makes the individual classes very hard to test in isolation. Reuse also becomes an issue. When ever `A` is used, it drags along its dependencies.

Apparently, there is a potential conflict in how resources traditionally are managed in C++ and DI. This will be further explored in *section 7.1.8 Dependency Injection and RAII.*

4. Case Study

The case study consisted of two major parts. Each conducted on separate code bases and in separate work processes:

- Part I operated on *Codebase I* and consisted of:
    - Bringing legacy code under unit test while noting impediments for test in isolation.
    - Identifying typical scenarios that trigger these impediments.
    - Identifying refactoring patterns that brings the code into compliance.
- Part II operated on Codebase II and consisted of:
    - Evaluating the IoC container PocoCapsule.
    - Evaluating the Dependency Injection Framework Autumn.

4.1. About the Code bases

In the following subsections a brief history of the two codebases used are presented. The description has in the case of *Codebase I* intentionally been left indistinct to comply with non-disclosure agreements.

4.1.1. Codebase I

It is important to recognize that the codebase provided by Enea has a long history and has not been under Enea's care throughout its entire lifetime. The state of the product, should therefore not be seen as a reflection of the current development teams practices but rather as a consequence of its long history

It originates more than 10 years back and was then written mainly in C for custom hardware. C++ was later introduced and is now the principle programming language for the codebase. It was then ported to the Windows platform and later on ported to use DirectX. The product has been owned by a number of different companies throughout its lifetime and worked on by equally, if not greater, number of different development teams. It is now owned by a company not disclosed in this report. Enea AB is for now responsible for maintenance and development.

The codebase's embedded C/C++ history dominates the overall program structure, design and architecture. Therefore, it was not suitable for test with IoC containers. Massive refactoring would be required to bring it to a state where dependency injection could be implemented for use with IoC containers. Additionally, the licensing terms for PocoCapsule prevented direct use in the production code. Any meaningful result would come at an effort beyond that of the academic project's scope as well as being less than meaningful from a production point of view. To have something to evaluate the IoC containers against, Codebase II was brought in.

4.1.2. Codebase II

As noted in the previous section, the codebase provided by Enea was not suitable for testing with the IoC containers during evaluation. In order to have something to evaluate against, this second codebase was used.

It is a small 2D game written by the author, in C++, for an introductory course in game development, taken at Blekinge Tekniska högskola a few years ago. The game uses HAAF's Game Engine (HGE)[12] to load textures and render graphics.

[12] HAAF's Game Engine product website: [93]

figure XVIII. Codebase II: Screenshots from the game

The game is a simple 2D top-down scroller with only a few components and textures. Even though far from polished, the original code structure in this codebase was far better suited for dependency injection than *Codebase I*. It is because of this it was used in the evaluation of the IoC containers.

4.2. Bringing Code Under Test

As recognized in *2.3 Investigative Method*, the case study was conducted without an authoritative meta-method. The heuristic method used for this part can best be categorized as a trial and error effort, iterating several times over a set of activities conducted on Codebase I:

- Unit Testing[13]
- Finding Impediments for Test in Isolation
- Scenario Identification and Pattern Discovery
- Refactoring

The activities are described in more detail in the following subsections. Even though the four activities are described separately, they were very much conducted hand in hand.

The entire process can be generalized into:

- Try to bring a class under test in isolation.
- Recognize failure to do so due to blocking factors such as global dependencies and hard coupling to other components.
- Identify those factors
- Note the situation that brought on the factors
- Remove them using dependency breaking techniques such as Inversion of Control, techniques from Michael Feathers' book [8] or by trial and error.
- Write tests that exercise the class in question.
- Fix any bugs found or report them.

4.2.1. Unit Testing

Unit testing was conducted with the help of the CppUnit framework. A separate test application was written and placed in its own project, in order to strictly separate test and production code. The test program made use of the CppUnit console test runner. The test runner printed the result to Visual

[13] Bugs found were either fixed or reported.

Studio output window, reporting the number of test run, passed and failed. The report format was customized to report failures in visual studio error format, allowing for click/link to source line for failed asserts.

The test program was made to execute as a post build event. That way the tests were always run after a successful build of the test suite. A visual studio macro was also written for easy switch between production and test configuration, making it possible to line step debug test runs.

Each component was tested in reasonable isolation with the use of stub objects as replacements for depended upon objects abiding to the methodology described in *3.5 Unit Testing*.

Tests were organized in class per class test-cases and class per fixture, following the xUnit patterns [25] (p. 156-157). Each test consisted of a number of small specific test cases. Using combinatorial testing [16] as much as possible, individual tests where designed to test a full concept or function of the component. For instance, one component had the ability to schedule objects to run frequently. One such concept would be to ensure that scheduled objects actually where run. Another concept would be to unschedule scheduled objects and verify that they actually were removed and that they only ran the correct number of times.

To imitate test first development, a technique here called *test inversion* was used. *Test inversion* is the practice of commenting out code, write a test and watch it fail. Uncomment parts of the code and verify that the expected tests succeeded.

A total of five test fixtures were written to test eight components, with a total of 78 tests. Eleven mock classes were written to help with testing in isolation. In the process, some additional sixteen adjacent classes where affected and required refactoring.

The aim of the unit tests was to gain enough confidence for more complete refactoring.

4.2.2. Finding Impediments for Test in Isolation

When a class was first brought under test it would always fail due to global dependencies or other hard coupling to other components. During the process of clearing these blocking dependencies, each was noted and characterized. The characterizations were later summarized into the impediments presented in *7.2 Impediments for Test in Isolation* as well as integrated with conclusions from existing sources.

The identification of the blocking factors was mostly done using compiler and linker error messages. Also the actual effort of writing tests for a class exposed problem areas.

4.2.3. Scenario Identification and Pattern Discovery

As discussed in previous section, when a class was first brought under test it would always fail. After the blocking factor was noted and characterized, the situation for when it occurred was noted and later summarized into a set of scenarios.

The actual process of removing the blocking factors took a lot of trial and error. An attempt was made, only to realize failure and revert to previous code state. Any situation where a change rippled through the entire application or otherwise causing cascades of changed was considered a failure. The aim was to do as small and safe refactorings as possible.

Safe in this sense should be taken as to mean that:

- few neighboring components were affected
- the process was transparent and its consequences predictable

This was repeated until a code state was reached where all blocking factors preventing tests in isolation had been removed. The successful path was transcribed and later summarized into refactoring patterns.

The result is discussed in section *7.6 Scenarios and Patterns*. *Appendix A* consists of a catalogue of scenarios and patterns identified.

4.2.4. Refactoring, Bug Fixing

In the process of writing unit tests, finding impediments for test in isolation, scenario identification and pattern discovery, general refactoring and bug fixing were done. This included:

- Splitting classes sharing the same header and cpp file into separate files.
- Extracting interfaces from classes depended upon by the classes under test.
- Rewriting the main application instantiation to be less procedural.
- Extracted functions and classes previously declared and defined in global common files into separate header and cpp files.
- Added performance counters in some key classes.
- Some stand-alone functions were pushed into the classes they operated on as new constructors or methods.
- Deleting dead code.

The refactoring was far from complete. It was limited to the vicinity of the classes brought under test and done with great care not to introduce new bugs. Some fifty files and classes were affected.

4.3. IoC Container Evaluation

In the second part of the project two containers/frameworks where evaluated. The IoC container PocoCapsule and the DI framework Autumn. They were selected because out of the few IoC containers/DI framworks available, they seems at a first glance most promising and where not tied to a specific component model. Qt IoC container was not considered due to its dependency on the Qt object model.

The process consisted of a small refactoring of Codebase II to bring it to a state where DI could be fully utilized using in-code factory/build methods. The codebase was then branched to allow incorporation of PocoCapsule and Autumn respectively. Codebase II had no unit tests so the evaluation could only be done with respect to use in a production setting.

The evaluation was quite limited, since the code used was not very large, nor complex. Other effects might be recognized in another setting and hence the conclusions made should be read in the light of this simple test.

The evaluation was done on the following criteria:

- Is inheritance from special classes required for the components injected, injected into?
- Is other tagging, marking of classes needed?
- What is the overhead in terms of C++ code when retrieving and using components from the container?
- Is the structure of the XML, easy to read, easy to find errors?
- Did it require a custom build step? If so, was it difficult to set it up?
- Did it add or reduce flexibility in terms of switching out components?
- Overall readability, did it add or reduce complexity?
- Overall impression, did it add or reduce complexity?

No formal method was used to evaluate these criteria, only the subjective view of the author.

4.4. Time Frame

The study was done by one person, the author of this report. Approximately 320 hours was spent on trying out different ways to refactoring code and writing tests. This does not take into account the time

it took to analyze the findings. Additional 80 hours was spent on evaluating the IoC containers. Total time spent on the project, including all the research done regarding: refactoring and testing techniques, the concepts of DI and IoC, IoC containers in general, testability, OOD, TDD; quality metrics, learning static analysis tools, test coverage tools and generating reports from those tools; analyzing and assemble all the results, and writing the final report approximate to about 1200 hours.

5. Tools and IDE used

This chapter describes the technologies used and their setup. The purpose of this chapter is to give a complementary understanding, in relation to chapter *4 Case* Study, of how the project was conducted.

5.1. Source Control

Subversion (v. 1.6.4) was used for source control. The repository was of an older model (v. 1.4) and did not support merge backs. As a consequence, a lot of the merging was done by hand. The TortoiseSVN client (v.1.6.7) from Tigris was used.

5.2. Visual Studio

Visual Studio, VS, is an IDE that used to be the standard for the Windows platform. However, alternative IDEs such as Eclipse has seen increased use the last couple of years. Visual Studio 2005 Professionals was used for the project related to Codebase I and hence it was also used for this project.

5.3. CppUnit

CppUnit is a unit testing framework, created by Martin Fowler and now maintained by Baptiste Lepilleur et al. It is a port of the popular JUnit, written by Kent Beck et al., a unit testing framework for Java, which in turn originates from SUnit written by Kent Beck, a testing framework for SmallTalk. CppUnit is released under the Lesser General Public License (LGPL), see [91].

CppUnit also exists in several lightweight versions, such as CppUTest (project website: [102]) which in turn is based on CppUnitLight. CppUnitLight is a minimalistic version of CppUnit created by Michael Feathers (project website: [88]). CppUTest is an extension of CppUnitLight tailored for embedded systems.

A separate test project, a test application, was written that used the CppUnit console test runner to run test suites. The test runner printed the result to the Visual Studio output window, reporting the number of tests run, passed and failed. The report format was customized to report failures in visual studio error format, allowing for click/link to source line for failed asserts. CppUnit version 1.12.1 was used.

5.4. Test Automation

Tests become really useful when they are automated. To achieve that, the unit test run was incorporated as a post-build event to a specific test build configuration. Hence, whenever the test-project was built, all the tests ran automatically.

5.5. Autumn Framework

Autumn framework claims to be a dependency injection framework, but it is actually an incomplete IoC container with a limited feature set.

It uses an XML-configuration to wire classes together but also requires macros to tag/markup the classes to create wrappers for the classes registered in the container. Alternately, the wrappers can be generated from the classes' header files using a command line tool. The project has not seen any development since July 2007 and is most likely dead. The documentation is sparse and refers only to only versions no longer in use. The Autumn framework can be downloaded from [86] and is released under Apache License 2.0 [85]. Version 0.5.0 of Autumn Framework was used.

5.6. PocoCapsule

PocoCapsule is close to a full-feature IoC container. It uses XML-configuration to wire classes together. It does not require additional markup of classes through inheritance or macros.

The XML-configuration feature is easily extended by the use of domain specific modeling (DSM). This is done by model transformation [105] using W3C Extended Stylesheet Language

Transformations (XLST) technology. This allows for high-level configuration, only requiring domain knowledge. The full implications of the exhaustive possibilities that the configuration engine offers, especially since it supports higher order transformation (HOT) [105] is outside the scope of this report. A full discussion on DSM in IoC frameworks and how it is implemented in PocoCapsule can be found at [98].

PocoCapsule can be downloaded from [97] and is released under LGPL [91]. Version 1.0 of PocoCapsule was used.

5.7. Code Analysis Tools

The C++ world lacks free code coverage tools for the Windows platform. At the time of the project, no free code coverage tools for the Windows platform could be found. Some obscure ones requiring cygwin setup, gcc and some additional toolset was available but was discarded due to complexity. Since it could not easily be integrated to the technical environment and current set of tools, none was used. As a consequence, all tests were written in the blind.

BullseyeCoverage version 7.13.13, from Bullseye Testing Technology (company website [87]), was used for the final analysis of test coverage but was not available until late in the project and could therefore not be used during the development phase.

Static code analysis tools for measuring code metrics was not made available until late in the project. They were only used to collect data at the very last stages. The tools used were QA C++[14] (product webpage [100]) from Program Research (company webpage [99]) and CppDepend version 1.2.1 (product webpage [90]).

Lattix LDM 4.8.1, from Lattix Software(company website [94]), was used to generate Dependency Structure Matrixes (DSM), initially intended to be used in the report. However, as it turned out, the tool was only used in the startup phase of the project to help with navigating and understanding the dependencies of Codebase I.

Proper profiling tools to get quality metrics and test coverage continuously would have been preferable and based on the experience from this project should be part of every programmer's tool-set.

[14] Versions used were: QA C++ 2.5, VS2005 integration 1.3.0, PDFReport 3.1, Baseline 1.2, Compiler Personality Generator 2.0.1, Compiler Wrapper 2.6

6. Status of the Codebase at Start

In an attempt to show the progress achieved after refactoring and testing, a set of quality metrics have been used to measure the state of the codebase provided by Enea, both before and after refactoring.

In this chapter, the state of Codebase I before refactoring is summarized. For a brief history and to understand how the codebase came to be, see section *4.1.1 Codebase I*.

6.1. Unit Testing

Before the project covered by this report started, no direct unit tests existed, no frameworks were in place. Hence the number of classes covered by unit tests were zero and consequently test coverage were also zero.

6.2. Application Statistics

The general application statistics are provided below. It is shown here so the overall progress of the application for the duration of the project can be taken into account. The experimental refactoring was done on a separate branch during normal product development and later merged with trunk. Hence the overall product evolved and hence changed the premise for the overall application statistics.

table I. Application Statistics: Codebase I, Before Refactoring

Number of lines of code:	62191
Number of lines of comment:	26454
Percentage comment:	29%
Number of classes:	317
Number of types:	481
Number of abstract classes:	23
Number of structures:	121
Number of templates:	2
Percentage of public types:	94.64%
Percentage of public methods:	86.86%
Percentage of classes with at least one public field:	17.11%

6.3. Quality Metrics

The next page displays a listing of the central classes worked on during the project. The names have been changed to obscure the product origin. Descriptions of the metrics collected can be found in *2.4 Quality Metrics*.

table II. Quality Metrics: Codebase I, Before Refactoring

Class	Rank	LOC	$LOCM_{max}$	$LOCM_{ave}$	LCOM	LCOM (HS)	CC_{tot}	CC_{max}	CC_{ave}	C_a	C_e	ABC	#M	#SM	#Fld	#SC	#BC
Database	11.29	1615	128	26	0.91	0.93	244	23	3.9	56	6	33	62	0	6	0	0
Engine	3.31	302	87	10	0.89	0.92	34	16	1.1	41	18	60	31	0	15	0	0
Language	0.43	171	53	19	0.44	0.5	32	13	3.6	2	7	19	8	1	3	0	0
Logger	21.26	80	13	8	0.57	0.63	8	2	0.8	117	3	15	10	0	3	0	0
Results	0.51	408	163	29	0.71	0.77	50	31	3.6	5	8	75	12	2	2	0	0

Rank: Type rank, a dependency measure based on Google PageRank algorithm
LOC: Lines of code
LOCM: Lines of code in method: max and average
LCOM: Lack of cohesion of methods of a class (CppDepend)
LCOM (HS): Lack of coehsion of methods of a class using Henderson-Sellers formula
CC: Cyclomatic complexity (CppDepend), class total, method max, method average
C_a: Afferent coupling
C_e: Efferent coupling
ABC: Association between classes
#M: Number of instance methods in class
#SM: Number of static methods in class
#Fld: Number of fields in class (member variables)
#SC: Number of subclasses
#BC: Number of base classes in inheritance tree

table III. Quality Metrics: Codebase I, Before Refactoring (cont.)

Class	DIT	NOP	NOC	WMC	RFC	LCM	CBO	ND_{max}	CYC_{max}	SPC_{max}	A	I
Database	0	0	0	73	275	1	18	8	22	1062	0	0.10
Engine	0	0	0	77	59	1	46	3	14	1149	0	0.31
Language	0	0	0	24	36	1	14	4	10	208	0	0.78
Logger	0	0	0	17	18	1	6	1	3	4	0	0.03
Results	0	0	0	22	60	2	27	7	30	21640	0	0.62

DIT: Depth in inheritance tree
NOP: Number of immediate parents
NOC: Number of immediate children
WMC: Weighted Methods per Class
RFC: Response For Class
LCM: Lack of Cohesion of Methods within a class (QAC++)
CBO: Coupling Between Objects
ND_{max}: Maximum nesting depth of control structures in a method.
CYC_{max}: Cyclomatic complexity (QAC++), method max
SPC_{max}: Static Path Count, method max
A: Abstractness
I: Instability

7. Results

This chapter discusses the results of the investigation. It will attempt to answer the research questions stated in *1.3 Objectives*:

Q1 What are the impediments for code to be testable in an isolated setting?

Q2 What are the steps for refactoring code to a testable state?

Q3 What are the benefits and drawbacks of the two IoC containers evaluated, with respect to the second code base, Codebase II, under investigation?

In addition to answering the explicit research questions, one section will be dedicated to presenting general insights learnt from the overall process.

The results are presented in a rather dogmatic, black and white perspective. However, the real world is painted in shades of grey and the suggestions should of course be adapted to contextual variations and conditions.

All statements made in this chapter are the author's personal opinions and based on the experienced gained from the case study –unless otherwise stated explicitly.

7.1. Lessons Learnt

The following subsections discuss a wide range of topics. Since the recommendations presented here do not always derive from collected data but rather are based on the general experience gained from the project, the result should be seen as just such –recommendations.

7.1.1. Test Project Organization

When organizing code it is important to separate concerns. One such separation should be that test code should not be mixed with production code [25](p. 28). When a product is shipped, it should be with absolute certainty that it only contains production code. Using defines and flags to switch between production code and test code have the potential for disaster[15] and can be very difficult to maintain[16]. As consequence, test code should be put in its own project.

If an application is well partitioned it will very likely at least contain some core functionality and a user interface, each contained within their own coding modules.[17]

Using the terminology of MS Visual Studio, the core and UI should reside in different projects, but within the same solution. A larger application might warrant further partitioning. A data intensive application should perhaps have its own data layer[18]. Each module should have its own set of unit tests. This conclusion is supported by how the NUnit's source and test code are organized. Every module is

[15] Accidently shipping test code can be difficult to detect and risk only surfacing from customers' thorough use. The main system may appear to be sound but one wrong set flag can trigger test code deep inside a component. In [25], "*Test Logic in Production*" is listed as a code smell: "*A system that behaves one way in the test lab and an entirely different way in production is a recipe for disaster!*" [25](p. 217)

[16] Define blocks clutters the code and forces double implementation production and test. This breaks the principle of single responsibility as well as making the code hard to read and understand; especially if the test code is sprinkled throughout a class.

[17] In [23], Martin gives application partitioning such importance that he dedicates an entire section on its principles (p. 251-384).

[18] In [10], Fowler lists *Presentation*, *Domain* and *Data source* as the principle layers of a layered architecture. Tiered architecture models such as MVC (model view controller), common in web applications, follows similar partitioning.

complemented with a corresponding test module holding all test code exercising the production module's code[19].

Roy Osherove, author of "the Art of Unit Testing"[20] and developer at TypeMock[21], even suggests having separate projects for unit tests and integration tests. This to avoid confusing their purpose and enable you to run all tests in an assembly and with confidence get all and only fast running unit tests. The developer shouldn't have to think about selecting a subset of tests. [34]

Following these practices one should end up with a setup similar to the one in *figure XIX*.

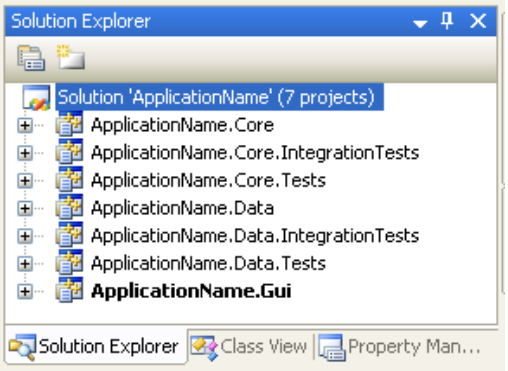


figure XIX. Test Project Organization

7.1.2. Application Partitioning

By partitioning the application into well-defined static libraries (libs), testing becomes much easier. Some components depend on other classes/structs that cannot or should not be mocked out for various reasons. Without the use of clear cut libraries, individual cpp files have to be included in the test project either as the class under test (CUT) or as a component to CUT.

It is much easier to include *simulation.lib*, *visualizations.lib*, etc. rather including all the individual cpp files. The overall dependencies become much clearer and the chances of a well defined set of interfaces to interact with the libraries become much higher. The partitioning helps reducing the overall coupling and hence reinforces a good architecture.

The downside of the library model is that it requires the lib to be in a buildable state at all times. Further, the average build and runtimes for the test suite risk running higher since the entire library needs to be built in order for a single class to be tested. Further, library-internal dependencies risk not being detected as easily as when pulling in individual classes. When working with individual classes, any dependencies, explicit or global, show themselves as link errors, if they are not defined in a header file. With static libraries, even library global dependencies do not break the linking stage.

However, if the ambition is to work properly with test driven development, nearly every class should at one point be tested and hence the test project risk becoming hard to maintain, managing all the production code classes as well as their corresponding test fixtures. In addition, when all production code classes are in the test project, the linker will fail to make dependencies explicit nonetheless since the depended upon classes are in the test project anyway.

The only problem is with library globals declared as the example below.

code example XVII. Global Declarations Using Extern

```
// globals.h
#include "component.h"
extern Component g_Component;
```

```
// globals.cpp
#include "globals.h"
Component g_Component;
```

[19] The source code for NUnit 2.5.3.9345 can be downloaded from [95]

[20] Complete reference at [84]

[21] Company website: [103]

Since globals.cpp will be included in the static library, it cannot be excluded in order to use the linker to detect unwanted dependencies. However, if all globals are collected in one file they are easy to spot and can be removed in the process of decoupling. If `extern` declarations are sprinkled throughout the code, they are bound to be included anyway along with classes being tested.

Based on the experience working with Codebase I, the conclusion becomes that partitioning the application into well delimited static libraries increases the chances of decoupled application architecture as well as making testing easier.

7.1.3. Build Times

The average build time can be reduced if classes are separated by interfaces/pure virtual classes. Refactoring an underlying class, by i.e. extracting method, does not force a complete rebuild since the interface doesn't change even if the implementation's header-file changes.

Time for a complete rebuild will most likely increase due to the increased number of files, but the average build time during development can be greatly reduced.

This becomes important in large projects. It is not acceptable to have a build time of 10-20 minutes just because a class gets a new private method. By following good principles for OOD, such as OCP, DIP and ISP (see section *3.11 Object Oriented Design)* the risk of so-called dependency magnets [57](time: 39:00 min) are greatly reduced.

Build times were not explicitly measured throughout the project, but the general experience was that the average build time was reduced as better separations and abstractions came into place.

At the time of project start, Codebase I had no separations through abstractions –i.e. central classes did not have interfaces. Added to that, central classes were distributed throughout the application as global variables using `extern` declarations.

The consequence was that even a small refactoring, such as changing the signature of a private method forced almost a complete rebuild. As interfaces were brought into place and global variables replaced with DI, build times could be greatly reduced.

Another problem area was that several classes shared the same header file and cpp file resulting in extended rebuilds just from changing one of the classes. As these classes were split up into individual files, the complete rebuild time went up marginally, from dealing with a greater number of files, while the experienced average build time went down thanks to reduced average file size.

7.1.4. Dependency injection: Preferred Methods

In section *3.8*, different methods for dependency injections were discussed. In complement, here follows a discussion on advantages and disadvantages of using these different methods.

The general consensus seems to be that constructor injection is preferable over other methods [40] [50] when it comes to applications, while setter injection might have its advantages for frameworks [54] [55].

Constructor injection is preferred, since it makes it possible to ensure complete objects. It also clarifies instantiation order, and exposes circular dependencies as illustrated by *code example XVIII*.

code example XVIII. Constructor Injection and Circular Dependencies

```
class Manager : public IManager
{
   class Manager(IFactory* factory);
};
```

```
class Factory : public IFactory
{
   class Factory(IManager* manager);
};
```

```
{
   // does not work, circular dependency
   IManager m = new Manager(f);
   IFactory f = new Factory(m);
}
```

This may very well indicate that a third class hides within the two classes.

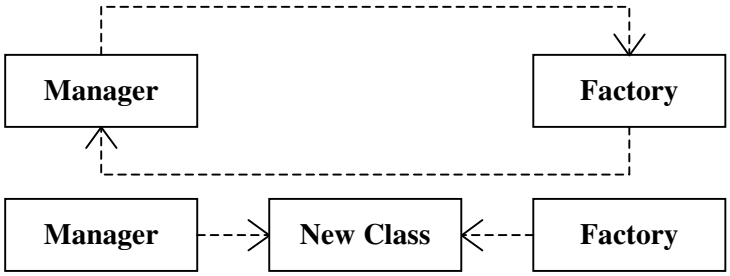


figure XX. Circular Dependencies Indicates New Class

Setter injection should be avoided since it can leave an object in an incomplete state. In a large application with a few hundred collaborators it can become difficult to ensure that all dependencies have been set. In contrast to constructor injection, static code analysis can usually not detect missing calls to setter methods. [40]

Another advantage of constructor injection is that any temporal coupling between the dependencies can be solved by the constructor. With setters, the knowledge of temporal coupling must be kept with the configuration component, rather than the object that actually causing the requirement.

However, setter injection is sometimes necessary. For instance, the use of object arrays, in C++, requires default constructors with no arguments.

code example XIX. Object Arrays and Setter Injection

```
C* listOfCs = new C[4];
for (int i = 0; i < 4; i++)
{
   listOfCs[i].setDependency(dependedUponObject);
}
delete[] listOfCs;
```

This can of course be mediated by using list of pointers or template injection.

Sometimes third party tools or frameworks might require setters and hence be a contributory factor to the style actually used in a project.

Another advantage of setter injection is that it allows for reasonable defaults. The constructor could provide the object with a reasonable, non-injected, default and hence make the object complete, while the setter allows for later specific configuration. This pattern could be useful since the IoC containers investigated, see section *7.8 Evaluation of Inversion of Control Containers*, require complete and detailed configuration.

However, default values have the disadvantage of requiring all dependencies to be included in the test project, even if they are later replaced by mock objects. If the proposed default object requires an expensive resource such as a database connection it is obviously not a good candidate. It would circumvent the whole point of IoC since the user would be depended on the existence of the resource even in the limited test scenario. See also section *7.1.6 Convention vs. Configuration*.

It also creates problem when it comes to lifetime management of object. See section *7.1.6 Convention vs. Configuration* and section *7.1.7 Lifetime Management* for continued discussion on issues regarding reasonable defaults and life-time management.

Both Arendsen in [54] and Smith in [55] argue that setter injection offers greater flexibility and is hence very suitable for frameworks and other constructs that require a lot of runtime interchangeability. The use of the *strategy pattern* [11] might sometimes warrant the use of setter injection.

Service locators, also mentioned in section *3.8*, might loosen the coupling between components but add another one instead, that of component to service-locator. The strength of IoC containers is that the components have no coupling to the container at all. In contrast, a service locator must be known by the component using its services. Jin, author of PocoCapsule, argues in [37], that Service locators has played out its part and are inferior to the more modern IoC container concept.

Templates in C++ offers a fourth possibility for Dependency Injection –namely Template Injection. In [11] it is suggested as a way to realize the strategy pattern. Templates offer the advantage of static code analysis, but without the overhead of an interface. Template injection is however limiting in that the injected component must abide by the implicit contract created by the component through its interaction with the template parameter type. If the injected type is an injectable in itself that require arguments to its constructor, the component also gets depended on the injected type. This creates a double directed dependency that negates the whole point of DI. In addition, template injection does not allow sharing of instances between two objects.

The following example illustrates the point.

code example XX. Template Injection and Implicit Interface

```
class Person;
```

```
class SQLDatabase
{
public:
   SQLDatabase(string connectionString);
   void write(const Person& person);
};
```

```
template<class DB>
class PersistantPersonnelManager
{
public:
   PersistantPersonnelManager(string connectionString)
      : m_db(connectionString)
   {}
   void save()
   {
      for(vector<Person>::iterator it = m_personnelList.begin();
            it != m_personnelList.end(); it++)
      { m_db.write(*it); }
   }
   ...
private:
   DB m_db;
   vector<Person> m_personnelList;
};
```

```
PersistantPersonnelManager<SQLDatabase> ppm("//admin@localhost:3090");
... //do some work
```

Notice how the connection string has to be supplied to the SQLDatabase's constructor in the initialization list in `PersistantPersonnelManager's constructor`. This creates an implicit dependency to the template parameter type's constructor signature. Also, the call to `write` on the database member creates the implicit contract of such a method signature on the template parameter type. This creates a second dependency, this time on the injected type.

7.1.5. Application Builders, Factory Methods and IoC Containers

IoC is mainly used to separate the concerns of instantiation, configuration object wiring and lifetime management to that of executing domain logic. For small applications, this results in a main method where all the components can be created on the stack and wired together and executed with a few lines of code. For complex applications, this might make main too large and hence make reading it and get an overview of it difficult. It is then suitable to extract some of the creation and configuration to factory methods.

For decently sized applications, it might even be warranted to create a separate class, an Application Builder, to encapsulate all the boilerplate code. The purpose of IoC containers existence is to simplify for the developer by abstracting all such code away and encapsulate it in the form of the container.

The use of IoC containers are illustrated in section *7.8 Evaluation of Inversion of Control Containers*. Below follows extracts from the builder class used in Codebase II prior to the use of IoC containers and shows manually written code that wires the application together. The main method instantiates the builder class, passing an instance of the Graphics Framework used to its constructor. After that, main calls the builder's factory method which assembles the game application and returns it to main. The code is far from perfect but illustrates the use of DI and application component wiring. *Appendix E* lists the complete source code for the `ApplicationBuilder` class.

code example XXI. Manually Wired Application

```
void main()
{
   HGE* gfw = InitGraphicFrameWork();
   ApplicationBuilder* builder = new ApplicationBuilder(gfw);
   if (gfw->System_Initiate())
   {
      _game = builder->buildGame();
      gfw->System_Start();
   }
   else
   {
      // If initialization failed show error message
      MessageBox(0, _gfw->getErrorMessage(), "Error", MB_OK | MB_ICONERROR
| MB_APPLMODAL);
   }
   delete _game;
   delete builder;
   ShutDownGraphicFrameWork(gfw);
}
```

```
class ApplicationBuilder
{
public:
   ApplicationBuilder(HGE* hge);
   ~ApplicationBuilder(void);
   Game* buildGame();
```

```
private:
   void LoadTextures(HGE* gfw);
   ScoreComponent* buildScoreComponent(HTEXTURE texture, const char*
fontFile);
   MenuComponent* buildMenuComponent(HTEXTURE texture, const char*
fontFile);
   HighScoreComponent* buildHighScoreComponent(HTEXTURE texture,
      const char* fontFile,
      const char* fontTitleFile,
      const char* highScoreFile);
   CreditComponent* buildCreditComponent(HTEXTURE texture, const char*
fontFile);
   ITerrainManager* buildTerrainManager(HTEXTURE tileTextureAtlas);
   LevelManager* buildLevelManager(HTEXTURE texture, const char* filename,
int startingLevel);
private:
   HGE* gfw;
   HTEXTURE terrainTexture;
   HTEXTURE scoreComponentTexture;
   HTEXTURE menuTexture;
   HTEXTURE backgroundTexture;
   HTEXTURE gameObjectTexture;

   ScoreComponent* scoreComponent;
   MenuComponent* menuComponent;
   HighScoreComponent* highScoreComponent;
   CreditComponent* creditComponent;
   LevelManager* levelManager;
   GameObjectFactory* gameObjectFactory;
   GameObjectManager* gameObjectManager;
   PhysicsComponent* physicsComponent;
   hgeFont* font;

   hgeSprite* scoreComponentBackground;
   hgeFont* scoreComponentFont;

   hgeSprite* menuComponentBackground;
   hgeFont* menuComponentFont;
   hgeFont* menuComponentFontShadow;
   hgeFont* menuComponentFontSelected;

   hgeSprite* highScoreComponentBackground;
   hgeFont* highScoreComponentFont;
   hgeFont* highScoreComponentFontSelected;
   hgeFont* highScoreComponentFontTitle;
   hgeFont* highScoreComponentFontShadow;

   hgeSprite* creditComponentBackground;
   hgeFont* creditComponentFont;
```

```
    ITerrainManager* terrainManager;
    ITerrainTileTypes* terrainTileTypes;
    ITileSprites* tileSprites;

    hgeSprite* levelManagerBackground;
    hgeFont* levelManagerFont;
    hgeFont* levelManagerFontShadow;
};
```

```
Game* ApplicationBuilder::buildGame()
{
    ...
    Game* game = new Game(gfw,
        terrainManager,
        scoreComponent,
        menuComponent,
        highScoreComponent,
        creditComponent,
        levelManager,
        gameObjectManager,
        gameObjectFactory,
        physicsComponent,
        cameraView,
        font);
    return game;
}
```

All this boilerplate code that is required to wire components together and manage components lifetime can be done using IoC containers. Their primary reason for existence is to help with this kind of configuration.

7.1.6. Convention vs. Configuration

As seen in *code example XXI*, in section *7.1.4 Dependency injection: Preferred Methods*, the wiring of components can be quite verbose. The lack of reasonable defaults forces extensive configuration. Spring for Java solves this by annotations. By annotating a class, it can be marked for being the default implementation when a specific interface is required. Spring resolves this using reflection.

The lack of reflection and class/method decorations forces C++ to be very explicit in its declarations and configurations of how the application should be wired together. A way for C++ to provide reasonable defaults could be implemented by declared default values in constructor declarations –see example below.

code example XXII. Default Value in Constructor

```
class Client
{
public:
    Client(IResource* r = new Resource()): m_r(r){}
    ~Client()
    {
        delete m_r;
    }
private:
    IB* m_r;
};
```

```
class Resource : public IResource
{
public:
   Resource(int i = 1): m_i(i){}
   ~Resource(void){}
private:
   int m_i;
};
```

An alternative route would be using null-checks in the constructor.

code example XXIII. Null-Check in Constructor

```
class Client
{
   Client(IResource* r == 0)
   {
      If (r == 0)
      {
         m_r = new Resource();
      }
   }
   ...
};
```

Null-check in constructor, however removes the option of passing in null for test-cases where the injected objected do not come into play. A so called *NullObject*, following the null object pattern, can of course be used but risk introducing other subtleties. If the NullObject is not managed properly, it risks propagating into sections not constructed to deal with such a construct resulting in unpredictable behavior.

Further, in both cases the explicit creation of the concrete class `Resource` makes `Client` less reusable. The use of `Client` requires the inclusion of `Resource` whether it will be used or not.

A third and perhaps the best option are overloaded constructors as in the example below.

code example XXIV. Overloaded Constructors

```
// client.h
#include “iresource.h”
class Client
{
public:
   Client(IResource* r);
   Client();
   ~Client(){delete m_r;}
private:
   IResource* m_r;
};
```

```
// Client.cpp
#include “client.h”
Client::Client(IResource* r)
   : m_r(r)
{}
Client::Client()
   : m_r(0)
{
   m_r = new Resource();
}
```

The header file does not need to expose the explicit dependency and hence leak it to other classes through the inclusion of header files. This could be a viable option if classes that are used together are packaged and deployed together as a cohesive component, such as a library. Since the entire package is deployed anyway, the inclusion of `Resource` doesn’t affect the perception and use of the library.

The downside of any use of default behavior is that it prevents instance reuse of the depended upon object. In *code example XXIV* the instance of `Resource` cannot be shared with other object instances, if the default behavior mechanism is deployed. Further is the issue of lifetime management. Ownership of the default object created has to retain with `Client`. As a consequence, injected dependencies also have to have their ownership transferred to `Client`, if not additional mechanisms such as the one in the example below are implemented.

code example XXV. Selective Life Time Management of Default Values

```
// client.h
#include “iresource.h”
class Client
{
public:
   Client(IResource* r);
   Client();
   ~Client(){ if (deleteR) delete m_r;}
private:
   IResource* m_r;
   bool deleteR;
};
```

```
// client.cpp
#include “resource.h”
Client:: Client (IResource* r)
   : m_r(r), deleteR(false)
{}
Client:: Client ()
   : m_r(0), deleteR(true)
{
   m_r = new Resource();
}
```

These kinds of mechanism however, are a cause of concern. The principle of lifetime management should not change depending on where the depended upon instance is created. Ownership should be clearly established, especially in an unmanaged language such as C++. If smart pointers such as `boost::shared_ptr` are used, this of course does not become an issue.

The experience from the case study confirms that reasonable defaults should be used sparsely and only if the three following conditions are met:

- All classes involved are packaged and deployed together.

- Default objects are inexpensive (do not require network resources, database connections etc).
- Automated lifetime management is utilized, such as the use of smart pointers or strong conventions on ownership.

Lifetime management is difficult in its own when working with unmanaged code such as most C++ and reasonable defaults brings another layer of complexity. Unless one is prepared for and has a clear strategy for this added complexity, a sometime extensive configuration is something that has to be accepted.

7.1.7. Lifetime Management

Generally speaking, any object passed as argument to a method or constructor should have the same or longer lifetime as the entity called. That means that an object passed to a method should still exist when the method returns[22]. This is an extension of the rules for injectables and newables suggested by Hevery depicted in section *3.10 Injectables and Newables*. Breaking this rule tend to complicate object management.

The following example demonstrates the point. Assume a component with a settings object. The settings object in turn is read from and written to a file. Further, the file needs a valid path in the file system to locate the file. See *figure XXI Lifetime Management example.*

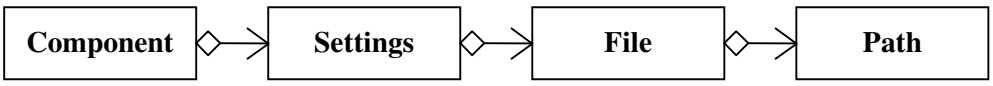


figure XXI. Lifetime Management example.

Following this, a small factory method assembles the component: it creates a path to the settings file on the file system and injects the File object with it. The File object in turn is injected in the settings object and so forth, and the factory method returns the complete component.

Following the rules previously established the File has a longer lifespan than the Settings object. This means that the settings object cannot close the file after reading the settings. It has to keep the file open, since it does not know about the path. This might be alright for some applications, but let us assume that the configuration file is read and modified by another application or the end user during the application's lifetime. If the file is kept open throughout the `Settings` object's lifetime, it will most likely be open for the duration of the application's run, preventing the user from editing the file.

Obviously the first attempt to class structure described initially does not work in this extended scenario. It breaks because the assumption of the File object's lifetime was wrong. The file's lifetime was not longer than the settings object and the proposed structure violates the lifetime rule. By restructuring as in *figure XXII*, Settings can open new files using Path, which does have longer life time than the settings object.

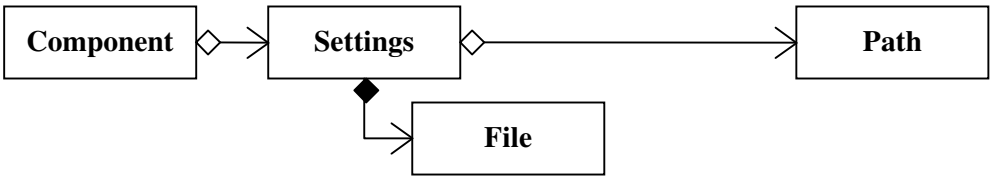


figure XXII. Lifetime Management example: Settings Owns File

[22] There are of course exceptions. If the method is part of the deallocation process of the object, the object should of course terminate before the method.

As established earlier, this change makes it harder to test Settings, since Settings creates the file itself. There is no way of replacing it with a mock, unless its use is abstracted through a factory. However, a factory will not be introduced here since the point of the example is to illustrate lifetime scope. In a real application with proper error-handling a factory would most likely have been warranted. To keep Settings testable, the process of reading and writing can be separated into several small functions exposing a suitable interface for testing.

code example XXVI shows how this can be done. Notice how the lifetime of the stream objects, (which are abstractions for files) are greater than the methods injected with them; hence abiding by the lifetime rule previously established.

code example XXVI. Lifetime Management: Exposing Testable Methods

```
class ISettings
{
public:
   virtual void read() = 0;
   virtual void write() = 0;
};
```

```
class Settings : ISettings
{
public:
   Settings(string& path) : m_path(path){};
   virtual void read()
   {
      ifstream reader = ifstream(m_path.c_str());
      read(reader);
      reader.close();
   }
   virtual void write()
   {
      ofstream writer = ofstream(m_path.c_str());
      write(writer);
      writer.close();
   }
   void read(istream& reader)
   {
      ...
   }
   void write(ostream& writer)
   {
      ...
   }
private:
   string m_path;
};
```

```
void testSettings()
{
   Settings s = Settings("fake path, not used");
   stringstream mockReader;
   stringstream mockWriter;
   s.read(mockReader);
   s.write(mockWriter);
}
```

Also notice how the `ISettings` interface preserves the abstraction from the Component's point of view. Even though `Settings` has extra public methods, the component doesn't need to know about them, since it is only deals with an `ISettings`.

`read(istream& reader)` and `write(ostream& writer)` could also have been made protected and called from a test-specific subclass.

Keeping track of all the injectable objects in an application can be burdensome. IoC containers can help with a lot of boilerplate code that manages the lifetime of the components. Creating manual build or factory structures to create and wire components together can be tiresome in large applications. Since with DI the components themselves no longer handle the creation and destruction of its dependencies, the build structure has to. For unmanaged languages such as C++, the references must be kept to all objects created, as they are needed in the clean-up process.

Objects injected into a constructor should have the same or longer lifetime than the newly created object. In contrast to other methods, the constructor should be considered to have the lifetime of the entire object, not only the constructor call. This is because of the complementary destructor, but also because the constructor is always called. It cannot be made virtual and hence cannot be overridden by subclassing. Doing work in the constructor is generally discouraged, see section *7.2.7 Work in Constructor*, so it should merely pass the injected object references along to be kept as member variables, and hence last at least the life time of object itself. If a reference passed in the constructor is not kept, it is usually of either doing work in the constructor or breaking the law of Demeter see section *7.2.8 Breaking Law of Demeter*.

7.1.8. Dependency Injection and RAII

The Dependency Injection design pattern is in some way in conflict with how RAII traditionally is implemented in C++. RAII stipulates that a dependency/resource should be created upon initialization and disposed upon destruction. This also includes the creation through copying an existing instance. Implementing RAII properly usually require a custom copy constructor and assignment operator in addition to complete constructor and destructor. With IoC the object itself no longer has that control. Hence, an object cannot know how to copy itself, since it does not know what exact implementation hides behind a referenced interface.

Valid copying could be implemented using specific clone methods, but since the IoC framework would not be the owner of such a created object, it had to be destroyed by the component itself, hence violating IoC.

Because of this dilemma, a consequence of IoC is that injectables cannot be copied or otherwise directly created outside the IoC container. Only the IoC framework knows how to create instances of injectables. To insure that copies are not made inadvertently the *copy constructors and assignment operators should be made private in injectable classes*.

The only way of creating new instances after the initial component configuration and wiring is through factories instrumented by the IoC framework or directly from the IoC container itself. The latter should be avoided except in the main method since this would create a dependency on the container; eliminating the main strength of IoC through DI compared to the use of Service Locators.

7.1.9. Dependency Injection and Unmanaged Code

Strictly speaking, unmanaged code usually refers to code that does not require a platform, framework or runtime environment such as JVM or CLR [30]. However, here unmanaged specifically refers to unmanaged memory. That is: the requirement to manually allocated and deallocated memory on the heap. Hence it implies the absence of an automated garbage collector mechanism.

After converting Codebase II from traditional RAII to constructor based dependency injection, it became clear that even though very powerful and probably the preferred way of handling dependencies, DI in combination with unmanaged code can become problematic. Using RAII, any shared resource can be clearly marked as such by passing it to the constructor, while private resources

are created directly in the constructor. With DI, it is difficult to establish a clean convention to convey ownership and distinguish between shared and private resources. It becomes especially troublesome with setter injection and the implicit convention of allowing for runtime substitution after initialization.

This matter has already been touched on in section *7.1.4 Dependency injection: Preferred Methods*, section *7.1.6 Convention vs. Configuration*, section *7.1.7 Lifetime Management* and section *7.1.8 Dependency Injection and RAII*, but it is so important that it is worth pointing out in its own section.

In managed memory environments this is not at all an issue. That can help explaining why DI has so much wider support in language communities supporting managed code.

7.2. Impediments for Test in Isolation

The following subsections discuss different factors that influence the concept that best can be described as a code attribute of testability. The purpose of identifying and describing these factors is because it helps to explain why a particular scenario is relevant from a testability standpoint. It also helps to pinpoint what in a pattern that makes it work, which factors that can be eliminated with easy refactoring and which are more dangerous and require more work.

7.2.1. Asserts in Test Paths

The way asserts are implemented on most platforms is that they either cause a complete halt of the program on fail, or do nothing. Neither is useful from a testability perspective. An assert in a test scenario should not halt the entire test run, only fail the current test and move on to the next test. Neither should it let an assert pass without action as this can cause null pointer exceptions[23] and other errors that cannot be recovered from.

Defensive programming can therefore hinder the possibility to exercise a class. At least if implemented with standard c asserts. When tests are used as part of the documentation, all aspects of a contract should be verified with tests and that includes that proper guard code is in place.

Away around this is using custom asserts such as the one defined in *Appendix A*: *Move Asserts to Where it Really Matters*. By throwing an exception instead of completely halting a program, unit testing frameworks such as CppUnit can detect triggered asserts, fail the test and move on to the next test.

Over-zealous use of guard code can hamper testing in other ways as well, i.e. by preventing the passing of null into the constructor. Passing null can be a valid test setup in white box testing if it is known that a specific subcomponent is not used when exercising a class in a particular way. Why not pass null to document this fact as well as making the test setup cleaner? If guard code checks for null in the constructor, this cannot be done. Instead, move the asserts to places where it really matters. If a method uses a specific member variable set through injection, why not assert in the beginning of that method instead of in the constructor?

If a subcomponent is critical, using references instead of pointers and null checks are always an option see *code example XXVII*. Programmers can of course work around it by passing a variable pointing to null, but using references sends a clear syntactic signal that null is not accepted.

[23] This assumes of course that the assert checks for null in a relevant manner.

code example XXVII. References Instead of Pointers and Null Checks

```
// pointers and null checks
class Client
{
public:
   Client(IService* service)
   {
      assert(service != 0);
      m_service = service;
   }
   ...
};
```

```
// using references
class Client
{
public:
   Client(IService& service) : m_service(service){}
   ...
};
```

7.2.2. Global State

Global state can be achieved in many ways. Some of the ways are in the form of:

- Global variables imported through an extern declaration in a widely used header file.
- Singletons, implemented through the traditional Singleton pattern in [11].
- Same object passed around through method calls or injected using DI or similar techniques.

The two first options cause problems when it comes to testing, making dependency injection the preferred method. Global variables are implicit dependencies that make the API of the class under test lie about its dependencies. Singletons are just global variables in disguise. Further, global state propagates between tests potentially causing tests to fail or pass depending on which order they are run. The most important aspect though, is that there is no enabling point for an object seam. Singletons and global variables cannot be replaced with test doubles using subclassing and polymorphism. The only way is using preprocessor or link seams, or by refactoring the code.

Many seem to share this view. Misko Hevery is one of many who have written extensively about the problems of Singletons and global state [45], [46], [47], [48]. Kent Beck also discourages its use in [1] (p. 179).

7.2.3. Stand-Alone Functions

Just as Singletons and global variables, stand-alone functions and class methods (static methods) cannot be overridden using subclassing and polymorphism. If the function in turn has implicit state because it uses function static variables, access global variables or Singletons, that dependency leaks over to any class that calls that function. Since stand-alone functions, an unfortunate legacy from procedural style of programming, do not offer any form of abstractions in its normal use, they become strongly coupled to any code that uses them.

7.2.4. Object Instantiation

Any object instantiation inside the application logic creates a strong dependency on that specific class as well as on the specific object created. This double dependency makes the code harder to test. The class cannot be replaced with a subtype for testing purposes. The object itself cannot be configured for

testing since it is locked away inside the class that uses it. Even if a reference to it can be obtained, the damage is already done. By instantiating a concrete class, that class has to be available at compile time. If class `A` instantiates an object of class `B`, `B` has to be included whenever `A` is used.

Further, since the instance of `B` is configured in `A` where it is created, it drags along any dependencies needed for that configuration. If B uses the network, connects to a server or accesses a database, those resources need to be available for `A` to compile and run. Even if `B` is later replaced with a test double, the resources still need to be made available for that initial setup. As a consequence, any object instantiation can be disastrous for testability.

Further, object creation inside a class could be considered a violation of the Single Responsibility Principle discussed in *3.11.1 Single Responsibility Principle*. By both being responsible for object creation as well as operating on it and performing tasks, the class does more than one thing. Object creation should be extracted and kept separate from the rest of the logic. The object graph should be created independently of its use.

Not surprisingly, Hevery has written extensive on this topic as well, [42], [43], [44], [45], and seems to share this view.

In summary, object instantiation mixed in with logic:

- Creates strong dependencies on concrete classes.
- Creates strong dependencies on specific instances.
- Violates Single Responsibility Principle by doing both object creation and performing tasks.

7.2.5. Complex Data

Complex data structures can hinder effective unit testing. Structs that has structs that has arrays of structs require complicated and error prone setup. Any function that operates on such a structure will be difficult to test. The combination of states that such a data structure can take in itself contribute to complexity. A consequence of this is that a massive amount of tests have to be written to take all different states into account, or risk low test coverage.

Complex data most likely suffers from other problems as well. It is usually a sign of that structure doing to much, having too many responsibilities, hence violating SRP. It is further impaired with poor encapsulation, exposing state, and as a consequence, any use will most likely result in a violation of the Law of Demeter.

In summary, complex data:

- Is difficult to setup.
- Has many possible states and therefore require extensive testing.
- Encourages bad design through improper placement of logic.
- Violates Single Responsibility Principle.
- Use results in violation of the Law of Demeter.

7.2.6. Exposure of State

Code that makes use of classes that expose state due to poor encapsulation can be difficult to test. Classes with exposed state are difficult to mock and encourages improper placement of functionality resulting in bad design. Code that uses classes with exposed state tend to directly operate on that state rather than ask the class to do it itself. This results in brittle tests since the exposed state has to be asserted against in the test which require knowledge of the inner workings of the function being tested. It also results in code duplication since the same operations are bound to be required by all that uses the badly encapsulated class, and hence that logic will be shattered throughout the application. Since use require manipulation of the exposed state, Law of Demeter violations are common.

In summary, classes with exposed state:

- Are difficult to mock.
- Forces brittle tests to be written, asserting against that state.
- Encourages bad design and shatters logic throughout the application.
- Results in Law of Demeter violations when externally modifying state.

7.2.7. Work in Constructor

Unlike other methods, the constructor always gets called and cannot be made virtual –hence cannot be overridden. Therefore, any kind of work except simple assignment should be avoided in the constructor. Conditional and looping logic can be difficult to test in a constructor and should therefore be avoided. Excess work can easily be extracted using the pattern *Extract Working Constructor*, in *Appendix A*.

Any use of the new operator, calls to static methods or external functions will inadvertently create unbreakable dependencies. Since a constructor neither can be overridden using subtyping, nor masked out by overriding wrapper methods, these kinds of dependencies cannot be broken within the scope of an object seam once created.

Basic types, such as the standard implementation of string, and class private types can usually be used without affecting testability. Private types are made private for a reason and should be completely hidden from the user of the class and can thereby be considered part of the implementation. However, if there exists a clean abstraction for the private class, consider making it public and treat it as any other dependency. The whole point of unit testing is to test small manageable pieces at a time. If private types and advanced internal data structures are making the class too big and complex to test with ease, consider extracting such dependencies to full-grown classes to reduce the complexity. The pattern *Extract Class* from *Appendix A* can be used for that.

However, an object should not be incomplete upon creation. Requiring additional values set or init methods called are indications that the object are incomplete and hence has hidden non-explicit dependencies and possible temporal couplings.

Removing all configuration of an object from the constructor, externalizing it, risk forcing the high level wiring of components to care too much about details. The IoC container specification file can become cluttered with tiny details about all the components' configuration that it risk obscuring the big picture.

This can be solved with specialization –see *figure XXIII*. By removing the entire configuration from the constructor, a generalized class can be created. The configuration can then be placed in a subtype, used in the production environment, wired in by the IoC container. The generalized class is easy to unit test while the specialization is easy to use with IoC containers and can presumable still be tested with integration tests. The pattern *Simplified Configuration Through Specialization* from *Appendix A* can be used for that.

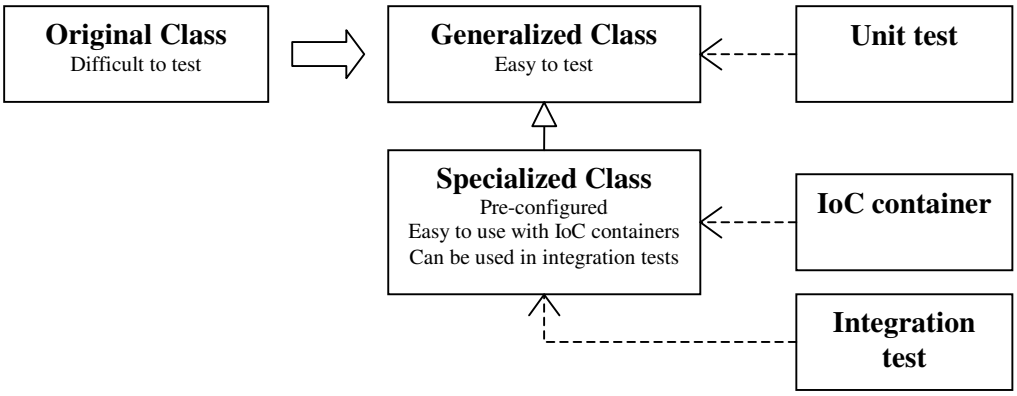


figure XXIII. Specialization Through Subclassing

Using a code example from Codebase II. The hgeFont is a class from HGE, the graphics framework used. The hgeFont class is a generalized class that allows for detailed configuration. All that configuration might not be relevant from the top level perspective. By subclassing the font, with specific configuration presets for each subclass, the overall intent can be come clearer.

code example XXVIII. Specialization Through Subclassing

```
class MenuItemFont : public hgeFont
{
public:
   MenuItemFont(const char* fontFilePath) : hgeFont(fontFilePath)
   {
      this->SetScale(0.8f);
      this->SetColor(ARGB(0xA0,0xAF,0xAF,0xA0));
   }
};
```

```
class SelectedMenuItemFont : public hgeFont
{
public:
   SelectedMenuItemFont (const char* fontFilePath) : hgeFont(fontFilePath)
   {
      this->SetScale(0.8f);
      this->SetColor(ARGB(0xEE,145,21,23));
   }
};
```

```
Class MenuItemShadowFont : public hgeFont
{
public:
   MenuItemShadowFont(const char* fontFilePath) : hgeFont(fontFilePath)
   {
      this->SetScale(1.0);
      this->SetColor(ARGB(0xA0,0x10,10,10));
   }
};
```

In summary, a constructor should:

- Not create its dependencies, except on basic and possible private types.
- Not call static methods.
- Not call stand-alone functions.
- Not make use of control flow such as conditional statement or loops.
- Result in a complete object.

Wolter et al. have reached comparable conclusions and has compiled a similar checklist in their guide to writing testable code [45].

Work in a class' constructor can in certain circumstances be warranted. In combination with RAII, it can sometimes be used to construct classes with similar properties to value objects; classes which require a lot of logic for configuration but once created only have values or very little operational logic.

7.2.8. Breaking Law of Demeter

Informally, the Law of Demeter (LoD) says that:

"*Each method can send messages to only a limited set of objects: to argument objects, to the self pseudovariable, and to the immediate subparts of self.*" [17](p. 38)

That means that a method may only call:

- Methods on objects passed to it as arguments.
- Methods accessed using the `this` pointer (self pseudovariable).
- Methods on member variables (sub parts of self).

This prevents calls to methods on objects that was returned by a method call, that in turn was returned by a method call, etc as in *code example XXIX*.

code example XXIX. Law of Demeter Violation

```
{
   return m_service.getComponentA().getSubComponentToA().Foo();
}
```

Violations of LoD results in the caller knowing too much about the inner workings of the callee. In the example, the caller has to know about Service, ComponentA and SubComponentA. This strong coupling results in three dependencies instead of just the one to the Service.

Misko Hevery lists Law of Demeter violations as one of the main flaws from a testability standpoint [49], [59](6:48 min). Classes should get what they need directly, and not traverse an object graph trying to locate it. Traversing the object graph creates dependencies to all the classes involved. Further, any mock object will need to implement the same object graph. Tests will be cluttered with setup code and fake classes, hiding the real intent of the tests; and these kinds of tests are brittle. Any change in any of the classes involved risk breaking the tests.

Law of Demeter violations can be sign of poor encapsulation. It can occur even with proper interfaces, but then it is usually a sign of improper placement of logic. The different classes' responsibilities will most likely need to be rearranged.

7.3. Unit Test Coverage

BullseyeCoverage, from Bullseye Testing Technology (company website [87]), was used measuring test coverage. All classes listed here did not have tests for all methods. This is reflected by the *Method # T/U* value (see *table IV Code Coverage During Unit Test Run* on page 71) which indicates the number of methods that was tested and not tested respectively. *Conditions #NE$_{test}$* shows the number of

condition outcomes not executed despite that the methods that they reside in were called. *Conditions* $\#NE_{tot}$ depict the total number of condition outcomes not executed.

When studying *table IV* on page 71, it is easy to realize that the sample is too small for any meaningful conclusions to be made. One can only note that unit tests have been written and cover a set of classes in accordance with directive D1.

If to speculate, the results suggest that the more complex classes were also difficult to test. Engine and Results are very complex (see *table XI* and *table XII*, *Quality Metrics: Codebase I, After Refactoring*) and also the classes with the least coverage.

The database class was unfortunately improperly tested. An entire set of method was left out by mistake. This was only discovered from the test report, but by then there was no time to go back and write more tests. Another interesting aspect of the database class is that it had a lot of internal null checks, actively preventing some states to be reached; explaining the high number of conditions not triggered by the tests.

If a reliable coverage tool had been available during the entire project, and the skills and knowledge accumulated throughout the project would have been available at the start, the number of classes covered by tests as well as the quality of the tests would have been much higher.

table IV. Code Coverage During Unit Test Run

Class	Cov (%)	Methods %	Methods # T/U	Conditions #NE_{test}	Conditions #NE_{tot}	Comments
Engine	52%	53%	23/20	22	38	Engine was later refactored further and better test coverage would most likely now be possible. All methods were not tested due to lack of time.
Loggable	100%	100%	12/0	0	0	New class and designed with testability in mind.
Logger	75%	100%	14/0	5	5	Some branches that required file system access was not tested.
Results	60%	64%	9/5	26	56	Complicated class with a lot of responsibilities. Operated on complex data structures with exposed state.
Database	58%	77%	47/14	158	239	A set of methods was unintentionally not tested. The miss was only discovered by the test coverage report and then it was not enough time to go back and complete the tests. Also, many branches were unreachable due to an over complicated schema of null checks.
RegList	100%	100%	1/0	0	0	Simple class tested as part of Engine.
SchedList	100%	100%	1/0	0	0	Simple class tested as part of Engine.
RemoveList	75%	100%	5/0	3	3	Class tested as part of Engine. Not all error cases tested.

Cov %: Total code coverage in percent. It does not take into account whether the code actually had explicit tests or not.

Methods %: Percentage of methods called during the tests.

Methods # T/U: Number of tested/untested methods.

Conditions #NE_{test}: Number of condition outcomes not executed in the methods tested.

Conditions #NE_{tot}: Total number of condition outcomes not executed in the class.

7.4. Code Readability

Little refactoring was done with regard to readability. Since the tools for regular checks of code coverage were not available the confidence level was initially not there to do any major refactoring. The changes required to bring the code to a testable state brought enough of complexity in many cases for anything else. Time was also a factor.

The skill set developed throughout the process would now allow for more aggressive refactoring, but was for obvious reasons not available at the time of the project.

The results presented here are to depict the changes in readability based on a set of code quality metrics. A complete description and motivation for the metrics used can be found in section *2.4 Quality Metrics*.

table V. Readability – Before Refactoring

Class	**LOC**	**$LOCM_{max}$**	**$LOCM_{ave}$**	**CC_{tot}**	**CC_{max}**	**CC_{ave}**	**WMC**	**ND_{max}**	**CYC_{max}**	**SPC_{max}**
Database	1615	128	26	244	23	3.9	73	8	22	1062
Engine	302	87	10	34	16	1.1	77	3	14	1149
Language	171	53	19	32	13	3.6	24	4	10	208
Logger	80	13	8	8	2	0.8	17	1	3	4
Results	408	163	29	50	31	3.6	22	7	30	21640

table VI. Readability – After Refactoring

Class	**LOC**	**$LOCM_{max}$**	**$LOCM_{ave}$**	**CC_{tot}**	**CC_{max}**	**CC_{ave}**	**WMC**	**ND_{max}**	**CYC_{max}**	**SPC_{max}**
IDatabase	40	1	1	0	0	0	1	0	1	1
Database	1627	128	26	245	23	3.9	76	**	**	**
IEngine	28	1	1	0	0	0	1	0	1	1
EngineBase	413	48	8	30	7	0.6	98	2	8	96
Engine	77	26	13	7	3	1.2	16	3	4	4
ILanguage	8	1	1	0	0	0	2	0	1	1
Language	171	53	19	32	13	0	25	**	**	**
ILoggedLanguage	3	1	1	0	0	3.6	3	0	1	1
LoggedLanguage	44	12	5	2	2	0.2	21	2†	2†	4†
ILogger	13	1	1	0	0	0	1	0	1	1
Logger	107	15	8	10	2	0.7	22	0	1	1
Loggable	59	7	5	0	0	0	16	0†	1†	1†
Results	412	161	29	50	31	3.6	25	7	30	21640

LOC: Lines of Code
LOCM: Lines of Code in Method: max and average
CC: Cyclomatic Complexity, class total, max and average
WMC: Weighted methods per class
ND_{max}: Maximum nesting depth of control structures in a method.
CYC_{max}: Cyclomatic complexity (QAC++), method max
SPC_{max}: Static Path Count, method max

Little noticeable change in readability can be detected from the collected metrics. Not surprisingly since almost no refactoring from readability perspective was done. The only noticeable change is in the Engine class that was split in two for future extension needs and was in the process cleaned up. The complexity has been greatly reduced. From a max cyclomatic complexity in any given method of

sixteen to seven using CppDepends measuring technique (CC_{max}) and from fourteen to eight using QAC++. The estimated static code path was reduced, from 1149 to 96. The longest method was also reduced from 87 lines of code to 48.

What has changed but cannot be detected by standard tools is that previously hidden dependencies now have become explicit.

table VII. Hidden and Explicit Dependencies Before Refactoring

Class	Hidden dependencies	Explicit dependencies
Database	1	0
Engine	6	0
LANG( )	1	0
Translate	1	1
ERROR( )	1	0
Results	2	0

table VIII. Hidden and Explicit Dependencies After Refactoring

Class	Hidden dependencies	Explicit dependencies
Database	0	1
Engine	0	7
M_LANG( )	0	1
LoggedLanguage	0	2
M_ERROR( )	0	1
Results	0	2

Hidden dependencies: the number of global variables, static and stand-alone methods used

Explicit Dependencies: the number of objects used that are passed as arguments to constructor, setter method or other injection path

The metrics have been collected manually and can be assumed to contain a degree of human error.

Only comparable classes, functions and macros are listed. The increase in dependencies for Engine is because a member variable, not counted as a hidden dependency, was changed to be injected for testability and hence became an explicit dependency.

LANG, and ERROR was global macro expansions that used global variables. Local variants were introduced instead, M_LANG and M_ERROR, that operated on member variables.

Due to the lack of good tools, and initial shortcomings with regard to skills, the code has improved very little in terms of suitable length of methods and method complexity. However, all dependencies have been made explicit hence increasing the readability from a relational perspective. It has become much easier to see what a class' collaborators are, and how it interacts with them. Further, readability on a file level has increased since classes previously sharing files were split into individual files. This was however not captured by the metrics collected.

7.5. Regression Testing

The investigated and refactored classes from Codebase I were tested in accordance with the product's test protocol and committed to project trunk in the source versioning system. The procedure for regression and product testing will not be discussed further since it is considered a trade secret. It will merely be stated that it was done, the tests passed and the changes are now integrated with the product.

7.6. Scenarios and Patterns

When setting out to refactor a complex application with a lot of tightly coupled classes and cross cutting dependencies, one has to realize that a complete refactoring of any given part will initially not be possible. This is especially true if the object graph is cyclic.

The old device "*fake it until you make it*" comes to mind and quite accurately describes the process. Do any single refactoring as complete as the system will allow. Eventually knots of interlocking dependencies will have been resolved and a short revisit will either allow for complete refactoring or another partial refactoring which in turn might allow further refactoring of other classes.

If possible, break the application into smaller pieces, making use of application partitioning described in *7.1.2*. Working on one piece at a time will help limit the scope and make things easier. Without limiting scope, refactoring risks trigger other refactoring needs, cascading into deep cross cutting changes that your limited tests cannot cash.

The identified scenarios for problematic code originate in violations of one or more of the impediments listed in section *7.2 Impediments for Test in Isolation*. Before diving into the different scenarios, each in its own subsection later on, a discussion on the nature of the patterns suggested is needed. It will explore the fundamental affects of different dependency breaking techniques.

7.6.1. Base patterns for refactoring

Through the case study, it has been found that refactoring done to break an unwanted dependency can be said to be of either of two types:

- Deep refactoring: weakens dependencies on both Meta level and base level.
- Shallow refactoring: only weakens dependencies on base level.

The techniques suggested in the consulted literature seem to share the same characteristics[24].

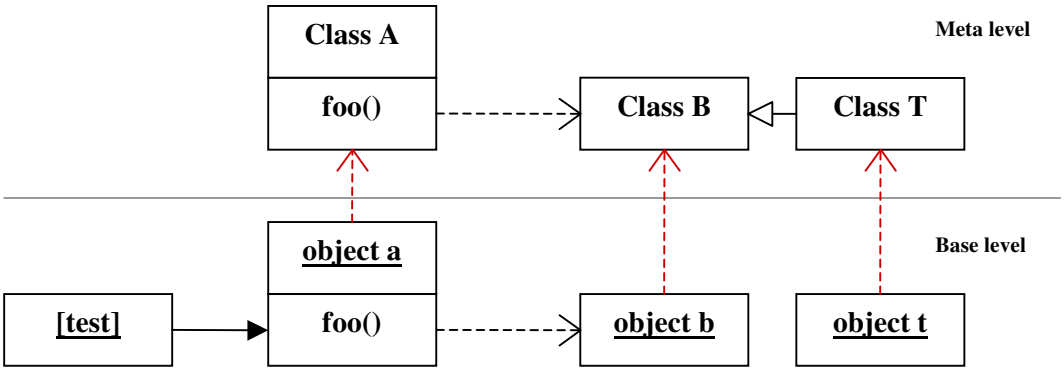


figure XXIV.Dependency structure before refactoring

For a test scenario such as in *figure XXIV*, where tests exercise an instance `a` (`object a`) of class `A` that has dependency to instance `b` (`object b`) of `class B` in method `foo`. No object seams exist to replace `b` with a test double `t`. Dependencies cross cutting from base level to Meta level (marked in red dependency arrows) is to signify that object `a` is of type `A`. All instances have an unbreakable dependency to its Meta object[25].

[24] A detailed study of all the patterns in [8], [9], [25] has not been done. A pattern by pattern categorization is outside the scope of this thesis. However, after reviewing the fundamental techniques suggested in these resources, it has not been possible to find anything to contradict the assumption stated.

[25] This is true for most languages, including C++. Dynamically typed languages that support complete reflection, such as Ruby, does not have this strong dependency.

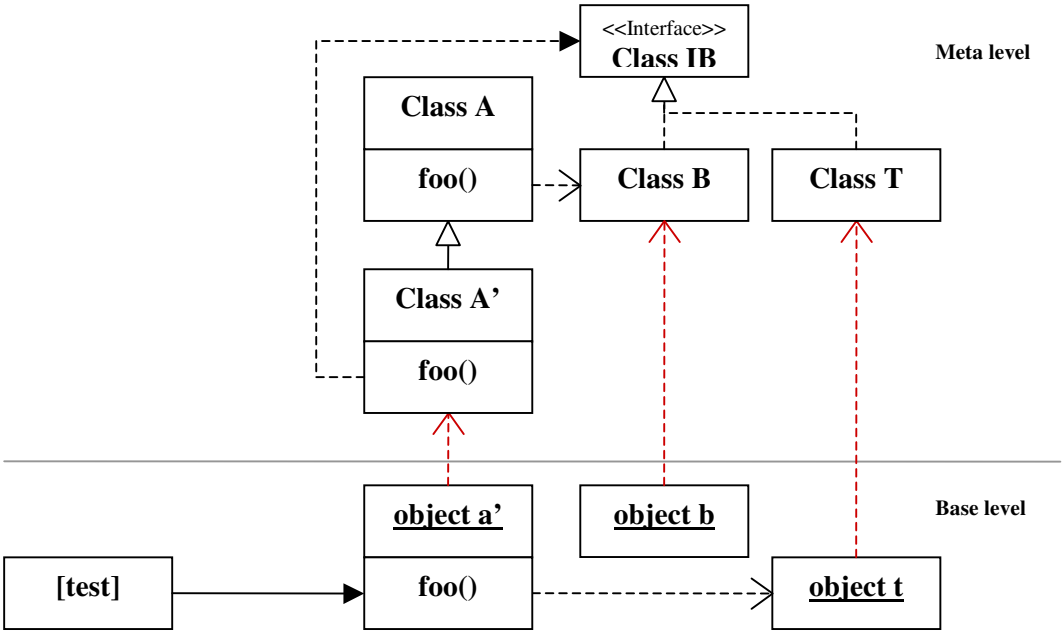


figure XXV. Dependency structure after applying shallow refactoring

After applying shallow refactoring on the situation, see *figure XXV*, instance `a'` (`object a'`) of class `A'`, where `A'` is a subtype of `A`, has dependency to instance `t` (`object t`) of `class T`, a test double for `B`, in method `foo`. An object seam has been created by overriding `foo` using subtyping and polymorphism. This eliminates the dependency on object `b` on the base level, but the dependency on the Meta level remains: `Class A` still depends on `class B`. `Class A` can now be exercised by the test through its subtype `class A'`.

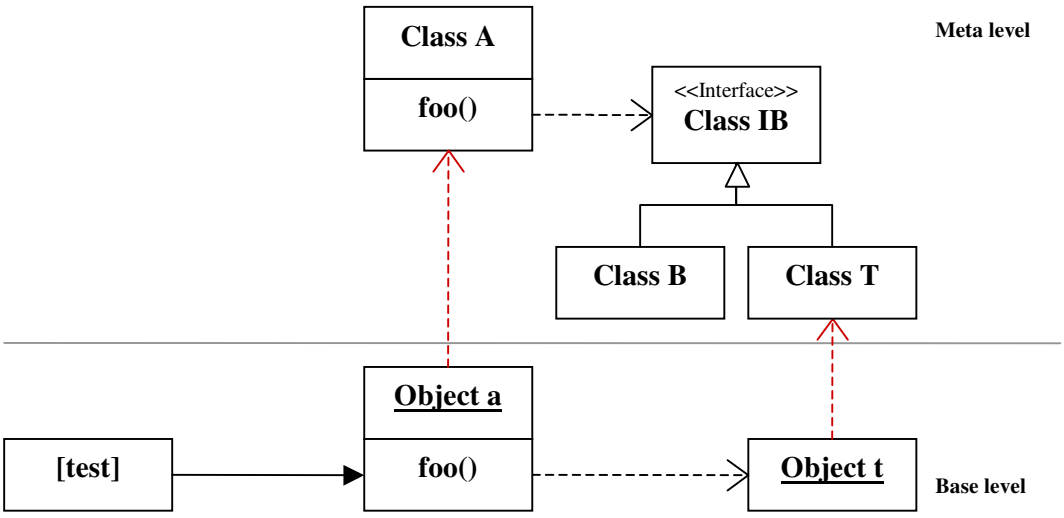


figure XXVI.Dependency structure after applying deep refactoring

If deep refactoring is applied, see *figure XXVI*, instance `a` of `class A` has no longer any dependencies except to the interface on the Meta level and to the test double `t` on the base level. The dependency on `instance b` as well as `class B` has been completely removed. This argument is based on the assumption that all dependencies are transient in the direction of the graph.

The distinction is important because even though shallow refactoring will improve the testability of a class in some cases, it will most likely not improve the design. The strong dependencies on the Meta level (dependencies to concrete classes rather than abstractions) will still prevent reuse since using the central class will automatically still require the presence of the depended upon classes even if it is

never used. The rationale of the shallow refactoring is merely to mask the problem with the distinct purpose of making a class testable. One should also note that the object B is still created in the case of the shallow refactoring. It is only its use that is masked. Hence, if B requires expensive resources to be instantiated, shallow refactoring will not remove that requirement.

The two types of refactoring reflect onto the type of patterns suggested. For any given scenario involving an unwanted dependency such as a global variable (global state transcends between test cases) or the direct reliance on a concrete class (prevents test in isolation), the dependency can be broken (only using object seams) in one of two ways:

- Complete removal through Dependency Injection, abstracted through an interface.
- Masking dependency, by isolating the use of the dependency in a virtual method that can be overridden using subtyping and polymorphism.[26]

There is actually a third kind of patterns. One not encouraged here. It is what could be called quasi-dependency injection where you replace an existing object with a friendly or a fake. It assumes that the type is virtual and the variable holding the object is protected (or has accessors) and of pointer type. One example of this is *Behavior-Modifying Subclass (Substituted Singleton)* in [25] (p. 586). It uses subclassing to add a method that substitutes the instance delivered by a Singleton[27]. The call to the static method getInstance creates a transitive dependency on the single instance. By replacing the instance using an implicit setter method introduced using subclassing it utilizes dependency injection, even though it is not done directly on the class that uses the instance and not in a very overt way. This is a shallow refactoring since the dependency on the concrete instance has been broken, but the Meta level dependency remains.

The proper utilization of Dependency Injection has two attributes that makes the practice significant:

- The removal of new in constructor, direct stack based member variable allocation or the use of global variables, removes the dependency on a specific object; hence breaking the base level dependency.
- Introducing an abstraction in the form of an interface to replace the dependency on the concrete class. This breaks the dependency on the Meta level.

If not both of these attributes are realized, the DI will not have the intended affect.

Dependency Injection can take the different forms discussed in 3.8 Dependency Injection (DI), but the basic elements of most refactorings implementing DI consists of these steps:

- Introduce field: to be used instead of the global reference.
- Change variable use: to use the local field instead of the global reference.
- Introduce parameter: in constructor or in setter method to make an injection path.

When masking dependencies, it mainly comes down to these basic steps:

- Encapsulate the dependency: extract its use to a method.
- Make the method virtual.
- Override the isolating method through subclassing.
- Mock implement the overridden method.
- Use the subclass for testing, the original class for production.

---

[26] Michael Feathers calls this "*Subclass and Override Method*" in [8]. It can also be done in the other direction, moving the core functionality to a base class and keeping the difficult dependencies in the original production class (now a subtype to the new base class). The base class can then be tested with ease. Feathers refers to this as "*Pushing Down Dependencies*".

[27] Singleton pattern in [11] (p. 127)

There are patterns of both types (DI and masking) in the catalogue of patterns presented in *Appendix A*.

7.6.2. Patterns

Appendix A contains a catalogue of refactorings/patterns to solve most of the problems described in the scenarios listed in the subsequent subsections. In order to make the scenario discussion efficient, refactorings applied are only referred to by name. A listing of the refactorings referenced is therefore presented here along with short descriptions. See *Appendix A* for a complete list and description of the patterns.

table IX. Refactoring Patterns

Name	Description
Removing Dependencies on Extern Declared Variable	The pattern illustrates how the dependency on a global variable declared with extern can be removed using DI.
Introducing Enabling Superclass	When many classes share the use of a component, instead of adding the field in every class, it is added through inheritance.
Force Strap a Component into a Test Harness: Refactory discovery	A method that can be used to discover what refactorings need to be performed. It helps discover which impediments that comes into play.
Inverting Control of Object Instantiation	Remove direct instantiation of dependencies in favor of DI and thereby allow for testing in isolation with the use of test doubles.
Extract Class	When logic manipulating a data-structure is duplicated and sprinkled throughout several classes. This pattern can be used to extract a class from the structure: encapsulating its members and removing the duplication by bring it into methods.
Extract Interface	Extract an interface from a concrete class to act as a barrier to prevent dependency leakage and help testing by creating an enabling point for an object seam.
Extract Locally Minimized Interface	Variant of Extract Interface, but only extracting a small set of the classes methods. Only those relavant for the class having the dependency -Abiding by the ISP.
Extract Working Constructor	Extract the work done in a constructor to a new class or factory method.
Move Assert to Where It Really Matters	Makes use of custom asserts, to not impede testing.
Simplified Configuration Through Specialization	Remove configuration from constructor by using DI, and after that making preconfigured instances available by subclassing the generalized class.

7.6.3. Scenario: Global Variables

The most common scenario encountered in Codebase I was the use of global variables through extern declarations, hence causing impediment 7.2.2. It was mainly the form of support components such as logger, language for translation of text strings and in-memory database working as an in memory repository for global values used throughout the application. But also other types of shared resources were used in this way.

This practice originates in the need for a small and predictable memory footprint when the code ran on custom hardware and had the properties of an embedded system. By allocating all components in the static memory area and accessing them through extern declared variables, the memory use became very predictable. However, this does not make for testable code in the more modern agile sense.

To resolve this, a refactoring spike was done. That is that a small set of classes was changed without affecting the entire codebase. The refactored classes were changed to utilize Dependency Injection to get their dependencies (using pattern *Removing Dependencies on Extern Declared Variable* from *Appendix A*). Interfaces were extracted from the dependent upon classes (using pattern *Extract Interface* from *Appendix A*). If it was a complex class used by many clients, a locally minimized interface (LMI) was extracted instead (using pattern *Extract Locally Minimized Interface* from *Appendix A*).

The statically allocated global variables was still used by the rest of the application, only now the changed classes got them using constructor injection instead of reaching out to the global space. This change made it possible to test the refactored classes in isolation and also made the dependencies more explicit. In the process of extracting interfaces on the depended upon classes, some logic was moved from the other code onto those classes (using pattern *Extract Class* from *Appendix A*) to make them more coherent and make the interface more reasonable.

7.6.4. Scenario: Unfriendly member variable

A common scenario was the use of unfriendly types as member variables associated through composition –violating rule about *Object Instantiation*, *7.2.4*. One such example was the EngineClock. A type used by the Engine class and in turn was dependent on Windows time –hence the operating systems real time. The clock was used to keep track of time durations and run times. Writing tests on something as brittle as real time are both hard and dangerous. It will most likely result in unruly tests with indeterministic behavior. It was obvious that the clock had to be replaced with a fake if the Engine class was to be tested properly.

The clock lacked interface so the solution started with extracting one (using pattern *Extract Interface* from *Appendix A*). After that, injecting it using constructor injection (see section *3.8.1*). The change made it possible to inject the Engine class with a fake clock instrumented to produce predictable time values.

7.6.5. Scenario: Work in Constructor Hidden by Init Method

The coding style of init methods originate from C where the only way of letting the programmer know that initialization had failed was by employing an init method that returned a status code. Init-methods are just extensions of constructors and any work done in an init method are just constructor work in disguise (7.2.7). The coding style also leaves with incomplete objects upon creation – the init method has to be called too. The only positive aspects of init methods are that they can be made virtual and overridden.

The refactoring spike ran into problems with regards to init methods. Some of the peripheral classes of the spike's scope made heavy use of init methods. One way of dealing with init methods is to extract them to a factory method (using pattern *Extract Working Constructor* from *Appendix A*). If it involves instantiation of member variables, the pattern *Inverting Control of Object Instantiation* can be used.

However, this assumes that the overall structure of the system has a good separation between object graph creation and domain logic. This was not the case here and extracting a factory method could therefore not be employed. At the time the problem could not be resolved due to other blocking dependencies outside the intended scope of the spike. The issue was therefore left unsolved and the spike took a slightly different direction to overcome the hurdle.

7.6.6. Scenario: Transitive Dependencies

A more devious scenario that came about was that of transitive dependencies. A class employed a macro or function that in turn had internal state or used global variables and thereby implicitly made

the class dependent on that as well. One such chain of dependencies encountered was the use of a global macro LANG that was an alias for a function call, Translate. The Translate function in turn was using two global variables for logging and translation (language).

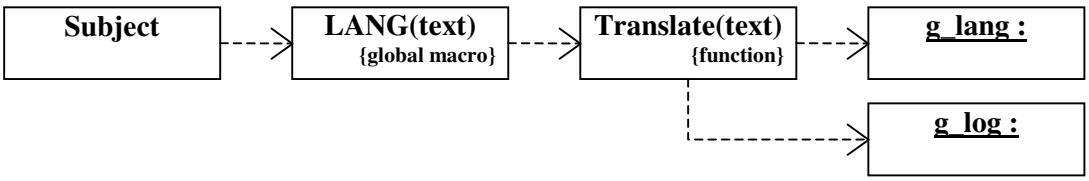
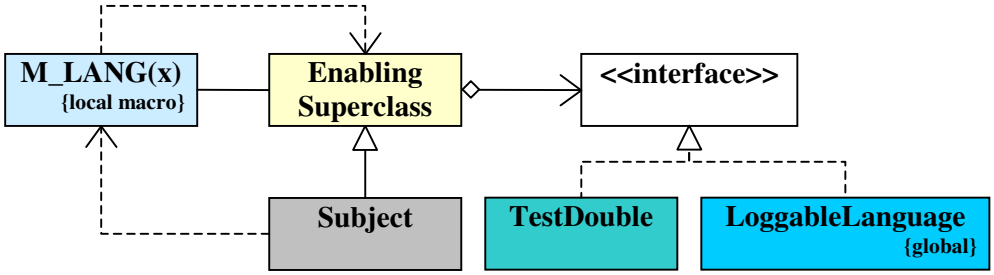


figure XXVII. Transitive dependencies

These kinds of chained dependencies can be broken where ever it is suitable. Usually it is better to break the chain as close as possible to the original class, since that is the link that requires the least widespread refactoring. In this particular case the structure was used all over the application and so the refactoring could not be too invasive.

It was solved by employing the pattern *Introducing Enabling Superclass* from *Appendix A*. The functionality of the `Translate` function was pushed to a subclass of `Language`. The global variable `g_lang` was changed to be of the new class `LoggedLanguage`. Since it was a true subtype of `Language` the rest of the application was not affected.



The new enabling superclass `Translatable` was introduced and a localized version of the macro was employed so the programming style was kept the same, but now, instead of accessing the global function, it used the methods of the superclass. The superclass' language field pointed to the global LoggedLangue instance so none of the functionality was changed. However, now when the class was instantiated for testing, a mock language object could be passed instead.

The choice of enabling superclass came out of the fact that since the structure was used throughout almost the entire application, many classes would eventually need to be refactored, if the global dependency is to be removed. The Enabling Superclass makes it easy to introduce the aspect.

7.6.7. Scenario: Poor Encapsulation

Some classes made use of global complex data structures that made them difficult to test. Besides having many possible states, the data structures were also difficult to imitate since they suffered from poor encapsulation. The situation had all the unfavorable elements of those described in sections *7.2.5*, *7.2.6* and *7.2.8*.

A way of solving this is to start by using the pattern *Extract Interface* (*Appendix A)* on the data structure. If the structure is large, the risk is that it is doing too much. Since it has been available, more and more have been attached to it. If this is the case it might be more suitable to use *Extract Locally Minimized Interface (LMI)* to start with. After refactoring all classes using the data structure, each with its own LMI, it might be possible to split the large class into several smaller; perhaps along the lines of the different interfaces.

The risk is that the class under test does a lot of manipulation of the data structure, since it has been public and very likely also violates Law of Demeter. At this stage the pattern *Extract Class* can be

employed to the class under test to make the depended upon data structure a little more cohesive and to make the interface a little simpler.

7.6.8. Scenario: No Abstractions

If no abstractions are in place, introducing tests becomes a slow process. Since dependencies can't easily be replaced with test doubles, the first act becomes to create an enabling point by *Extract Interface* (*Appendix A)* on all the depended upon classes, followed by modifying or adding constructor for dependency injection on the class under test; in practice changing the association between classes from composition to aggregation.

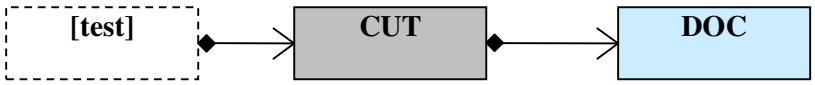


figure XXVIII. Association Using Composition

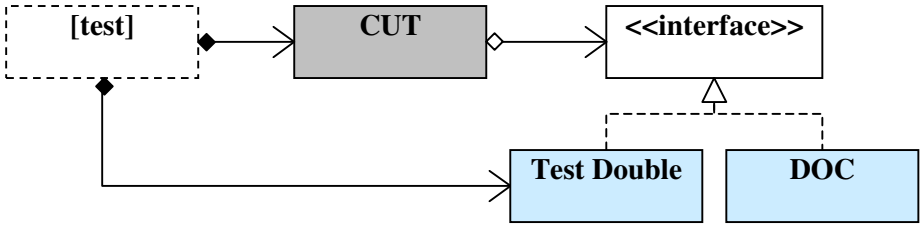


figure XXIX.Association Using Aggregation

This was often the case when writing tests for classes in Codebase I. Most classes were optimized for stack use –only concrete member variables. It appeared as though almost nothing was allocated on the heap. This has the advantage of making object creation very fast and lightweight since you don't run the risk of heap defragmentation. However, the implemention made the application very brittle since concrete stable classes, that needed to be instable, relied on other concrete stable classes.

7.7. Validation of Results

The primary aim of the refactoring was to bring previously unwillingly classes to a testable state. Through this process that involved eliminating factors identified in *7.2 Impediments for Test in Isolation*, it was hopped that design would have improved also. The hope was that after refactoring, the code would to a greater extent, follow the design rules presented in *3.11 Object Oriented Design and Its Effect on Testability*. In an effort to verify an increase in code quality and thereby validate the effectiveness of the patterns, a set of metrics were collected before and after the refactoring. In this section, an attempt to interpret the results is made.

As seen from *table X*, Codebase I grew about 25% as part of normal product development during the investigation. However, due to the limited sample of classes refactored, any statistical analysis would hold very little value any way. So the change of conditions has little impact in the conclusions discusses later in this section.

A larger sample size would have been preferred, but the project's time frame did not allow for more. The discussion that follows therefore lacks any statistical significance and should only be seen as an indication of progress rather than direct proof.

table X. Application Statistics: Codebase I, After Refactoring

Number of lines of code:	78550	+26%
Number of lines of comment:	34411	+30%
Percentage comment:	30%	+3%
Number of classes:	390	+23%

Number of types:	585	+22%
Number of abstract classes:	45	+96%
Number of structures:	137	+13%
Number of templates:	4	+100%
Percentage of public types:	94.74%	+0%*
Percentage of public methods:	84.28%	-3%*
Percentage of classes with at least one public field:	14.6%	-15%*

** The change is calculated the same way as for the rest of the values. It is expressed as a relational change in percent. Not in unit percent as easily mistaken when it comes to percentages.*

table XI. Quality Metrics: Codebase I, After Refactoring

Class	Rank	LOC	$LOCM_{max}$	$LOCM_{ave}$	LCOM	LCOM (HS)	CC_{tot}	CC_{max}	CC_{ave}	C_a	C_e	ABC	#M	#SM	#Fld	#SC	#BC
IDatabase	25.56	40	1	1	0	0	0	0	0.0	8	1	0	41	0	0	1	0
Database	6.94	1627	128	26	0.91	0.93	245	23	3.9	74	9	35	63	0	6	0	1
IEngine	3.09	28	1	1	0	0	0	0	0.0	45	3	0	29	0	0	2	0
EngineBase	0.22	413	48	8	0.93	0.95	30	7	0.6	1	23	65	49	0	18	1	1
Engine	0.15	77	26	13	0	0	7	3	1.2	0	9	21	6	0	0	0	2
ILanguage	2.05	8	1	1	0	0	0	0	0.0	3	3	0	8	0	0	3	0
Language	0.29	171	53	19	0.44	0.5	32	13	0.0	1	9	20	8	1	3	1	1
ILoggedLanguage	1.5	3	1	1	0	0	0	0	3.6	7	1	1	3	0	0	1	1
LoggedLanguage	0.38	44	12	5	0	0	2	2	0.2	2	6	10	9	0	0	0	3
ILogger	8.75	13	1	1	0	0	0	0	0.0	15	0	0	14	0	0	1	0
Logger	18.39	107	15	8	0.67	0.72	10	2	0.7	122	4	18	14	0	3	0	1
Loggable	1.79	59	7	5	0.5	0.55	0	0	0	10	1	5	12	0	1	8	0
Results	0.45	412	161	29	0.76	0.82	50	31	3.6	5	9	78	12	2	3	0	1

Rank: Type rank, a dependency measure based on Google PageRank algorithm
LOC: Lines of code
LOCM: Lines of code in method: max and average
LCOM: Lack of cohesion of methods of a class (CppDepend)
LCOM (HS): Lack of cohesion of methods of a class using Henderson-Sellers formula
CC: Cyclomatic complexity (CppDepend), class total, method max, method average
C_a: Afferent coupling
C_e: Efferent coupling
ABC: Association between classes
#M: Number of instance methods in class
#SM: Number of static methods in class
#Fld: Number of fields in class (member variables)
#SC: Number of subclasses
#BC: Number of base classes in inheritance tree

Experimental refactoring was done at a late stage where all use of the database variable and logger variable were changed to be of type IDatabase and ILogger instead. This resulted in the changes presented below: This was done after the academic license for some of the tools had expired and could therefore not be done as part of the overall results.

IDatabase: Rank=39.9, Ca = 81, Ce = 1, I=0.01
Database: Rank = 0.15, Ca = 0, Ce = 9, I=1
ILogger: Rank = 21.91, Ca = 136, Ce= 0, I=0
Logger: Rank=0.27, Ca=1, Ce=4, I=0.80

table XII. Quality Metrics: Codebase I, After Refactoring (cont.)

Class	DIT	NOP	NOC	WMC	RFC	LCM	CBO	ND_{max}	CYC_{max}	SPC_{max}	A	I
IDatabase	0	0	1	1	1	0	0	0	1	1	1	0.11
Database	1	2	0	76	277	1	18	**	**	**	0.02	0.11
IEngine	0	0	1	1	1	0	0	0	1	1	1	0.06
EngineBase	1	3	1	98	79	1	45	2	8	96	0.02	0.96
Engine	2	1	0	16	12	2	6	3	4	4	0.17	1.00
ILanguage	0	0	2	2	2	0	0	0	1	1	1	0.50
Language	1	1	1	25	36	1	14	**	**	**	0.11	0.90
ILoggedLanguage	1	1	1	3	2	0	0	0	1	1	1	0.13
LoggedLanguage	2	3	0	21	11	1	0	2†	2†	4†	0.11	0.75
ILogger	0	0	1	1	10	0	0	0	1	1	1	0.00
Logger	1	1	0	22	24	1	6	0	1	1	0.07	0.03
Loggable	0	0	7	16	12	1	4	0†	1†	1†	0.08	0.09
Results	1	1	0	25	61	2	27	7	30	21640	1.00	0.64

DIT: Depth in inheritance tree
NOP: Number of immediate parents
NOC: Number of immediate children
WMC: Weighted Methods per Class
RFC: Response For Class
LCM: Lack of Cohesion of Methods within a class (QAC++)
CBO: Coupling Between Objects
ND_{max}: Maximum nesting depth of control structures in a method.
CYC_{max}: Cyclomatic complexity (QAC++), method max
SPC_{max}: Static Path Count, method max
A: Abstractness
I: Instability

*** The tool used to collect these metrics silently failed to analyze these classes and it was not discovered until after the academic license had expired. Since these classes only had minor refactoring done to them it is not of great importance.*

*† These values were calculated by hand because of the failure mentioned in **.*

As stated before, the sample size for the classes structurally refactored strictly in accordance with the patterns outlined in *Appendix A* is too small to hold any statistical significance. However, an educated guess based on the metrics collected shows a slight improvement in both design and testability.

Prior to refactoring, the `Database`, class had a rank of 11.29, abstraction of 0 and Instability of 0.10 making it a concrete stable class central to the application (data from *table II* and *table I*). The class had all the attributes of rigidity according to Martin's definition in section 3.11. All the classes: `Database`, `Engine`, `Language`[28], `Logger` and `Results` suffered from the same problem –concrete stable classes depending on concrete stable classes.

After refactoring, this had changed. The application's dependencies on database and logger were reduced in favor of using their interfaces. This became particularly clear from an experimental refactoring done late in the project. By changing the types of Database and Logger to their respective interfaces, for all their use, the abstract stable classes became central while the concrete instable classes had almost no couplings at all[29].

Beyond the complete refactoring of Logger and Database, the other classes listed in *table XI*, *table XII*, as well as many other classes affected by the refactoring but not listed in the table, hardly showed any change at all in terms of metrics. The problem with measuring was that the refactoring was done on a relatively small portion of the application, while the measurements could only be collected on the application as a whole. The potential changes from the refactoring were too small to be noticed in the noise of the rest of the application.

When studying individual classes some general improvements can be seen:

- Introducing interfaces between concrete classes brought the code closer to adhering by the Open Closed Principle (OCP, section *3.11.2*).
- By removing instantiation in the forms of using new in the constructor and inlined concrete members (impediment *Object Instantiation*), multi responsibility (both object creation and performing tasks) was reduced to something closer to Single Responsibility Principle (SRP, section 3.11.1). Object instantiation also prevents extensions; hence violating OCP. It also violates Dependency Inversion Principle (DIP, section *3.11.4*).
- Relying on global state (impediment *Global State*) and stand-alone functions (impediment *Stand-Alone Functions*) violates OCP, since they are not extendible, as well as DIP due to how the dependency is acquired.
- Complex data structures (impediment *Complex Data*) have too many possible states and hence violate SRP.
- Exposing state (impediment *Exposure of State*) breaks abstractions barriers and hence does not adhere by OCP.
- Work in constructor (impediment *Work in Constructor*) is usually a sign of SRP violations and since constructors are none-virtual it also potentially violates OCP.
- Breaking Law of Demeter (impediment *Breaking Law of Demeter*) makes dependencies leak and thereby preventing effective reuse, hence violating OCP. It can also be a sign of a class that is doing too much, hence should be concerned about SRP.
- The pattern *Removing Dependencies on Extern Declared Variable* makes use of OCP and DIP.

---

[28] The use of the Language class and later LoggedLanguage was masked by a global function and therefore the obtained ranks do not do the classes justice. Unfortunately, the tool used did not provide a measurement for stand-alone functions.

[29] *IDatabase: Rank=39.9, Ca = 81, Ce = 1, I=0.01* *Database: Rank = 0.15, Ca = 0, Ce = 9, I=1*
ILogger: Rank = 21.91, Ca = 136, Ce= 0, I=0 *Logger: Rank=0.27, Ca=1, Ce=4, I=0.80*

- The pattern *Introducing Enabling Superclass* is just an extension of *Removing Dependencies on Extern Declared Variable* and hence also relates to OCP and DIP.
- The pattern *Inverting Control of Object Instantiation* makes use of DIP and SRP.
- The pattern *Extract Class* and *Extract Interface* brings the code closer to abiding by OCP.
- The pattern *Extract Locally Minimized Interface* makes use of *Interface Segregation Principle* (ISP, section *3.11.5*)

The effectiveness of the patterns in regard to design principles could only be validated in part. The metrics collected do not directly indicate an improvement due to the noise level of the rest of the application. In addition, the metrics themselves do not directly measures compliance to any one principle.

Further, the testability measures suggested by the referenced literature were not enough to justify some of the impediments found. The use of global state, procedural programming and misuse of asserts cannot be measured with the metrics suggested. Apparently the current principles for software design and the metrics for measuring testability is not complete enough to trigger on actual none-testable code. Practical testability is not completely captured in today's design principles and code quality metrics.

It can also be noted that shallow refactoring does not solve the inherit problem of dependencies on the Meta level. It is only a cheat to bring code to a testable state.

The SOLID principles are mainly about OOD, which in terms primary goal is to manage dependencies. Since strong base and Meta level dependencies are the primary impediments for testable code, the refactorings that bring code closer to SOLID will also make the code more testable.

Khan and Mustafa points to the central aspects of testability in [14]:

"*Achieving testability is mostly a matter of separation of concerns, coupling between classes and subsystems and cohesion.*" [14] (p. 1)

The problem of capturing unwanted dependencies in a suitable metric prevents proper validation of any practice that regards to refactoring, OOD and testability

7.8. Evaluation of Inversion of Control Containers

The use of IoC containers are mainly to help with the factors that are inverted with IoC:

- Object Instantiation
- Object Configuration
- Object wiring
- Lifetime management

All this can be managed by manually written code, but IoC containers simplify matters by eliminating the need to write the boiler plate code that do all these things. Instead, IoC containers use an complementary notation, often in the form of an XML configuration file, to specify the constraints. In the following subsections, two IoC containers are evaluated. As part of the evaluation they were used to wire an application together, which code as been denoted Codebase II and described in more detail in *4.1.2 Codebase II*.

Below follows a small part of the boiler plate code originally needed to wire and configure the application. The complete listing of the `ApplicationBuilder` can be found in *Appendix E*. Analogous IoC container configuration will be given in the respective subsections along with the evaluation.

code example XXX. Manual Wiring of Game Application

```
Game* ApplicationBuilder::buildGame()
{
   LoadTextures(this->gfw);

   Rect cameraView(Rect(0.0f,0.0f,600.0f,600.0f));

   tileSprites = new TileSprites(terrainTexture,
      60.0f, 60.0f, 2.0f, 3.0f);
   terrainTileTypes = new TerrainTileTypes(tileSprites);
   terrainManager = new TerrainManager(terrainTileTypes);

   scoreComponent = buildScoreComponent(scoreComponentTexture,
      "fonts/BankGothic_Lt_BT_24.fnt");
   menuComponent = buildMenuComponent(menuTexture,
      "fonts/BankGothic_Lt_BT_64.fnt");
   highScoreComponent = buildHighScoreComponent(
      backgroundTexture,
      "fonts/BankGothic_Md_BT_20.fnt",
      "fonts/BankGothic_Lt_BT_64.fnt",
      "highscore.txt");
   creditComponent = buildCreditComponent(backgroundTexture,
      "fonts/BankGothic_Lt_BT_20.fnt");
   levelManager = buildLevelManager(backgroundTexture,
      "levels/level.lev", 1);
   gameObjectFactory  = new GameObjectFactory(gameObjectTexture);
   gameObjectManager = new GameObjectManager(gameObjectFactory);

   physicsComponent = new PhysicsComponent();

   font = new hgeFont("fonts/BankGothic_Lt_BT_20.fnt");
   font->SetScale(1.0f);
   font->SetColor(ARGB(0xFF,0xFA,0xFA,0xA6));

   Game* game = new Game(gfw,
      terrainManager,
      scoreComponent,
      menuComponent,
      highScoreComponent,
      creditComponent,
      levelManager,
      gameObjectManager,
      gameObjectFactory,
      physicsComponent,
      cameraView,
      font);
   return game;
}
```

7.8.1. Autumn Framework

Autumn framework, claiming to be a dependency injection framework, but are actually an incomplete IoC container if this reports definition is to be used. Incomplete in the sense that it lacks some of the core features expected from an IoC container.

The evaluation was done using the latest version which was 0.5 released in July 2007. The framework has seen no further development since. It has no working community with regard to message groups.

All information and documentation are in simplified Chinese. Most of the documentation is also available in English, most likely generated from a translation service, based on the quality of the language. The information available however, refers to an older version and is completely inaccurate with regards to the current version.

After spending several hours dissecting the source code the basic functionality was discovered. All findings presented here are based on self discovery. Features might be available but hidden in the source.

The basic principles of Autumn are the use of a xml configuration files and a bean factory created from a static function call. The XML describes the relationship between beans, using a simple structure. These beans represent objects that get wired together and returned upon request.

code example XXXI. Autumn Bean Factory

```
Autumn::IBeanFactory* bf = Autumn::getBeanFactoryWithXML("file.xml");
IService* service = bf->getBean("myService");
service->doStuff();
bf->freeBean(service);
Autumn::deleteBeanFactory(bf);
```

```
<?xml version="1.0" encoding="UTF-8"?>
<autumn xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="autumn.xsd">
   <library path="local">
      <beans>
         <bean class="MyService" name="myService">
            <properties>
               <property name="NumberOfTheads">
                  <value>4</value>
               </property>
            </properties>
            <constructor-arg>
               <argument>
                  <ref bean="some-other-bean">
               </argument>
            </constructor-arg>
         </bean>
      </beans>
   </library>
</autumn>
```

The `property` attribute follows the Java model for properties and assumes the existence of `getPropertyName` and `setPropertyName` methods. This design choice is unfortunate since this enforces a specific naming.

The library tag either specifies "local" for self referencing the running assembly, or the path to an external dynamically linked library (dll).

The factory uses wrapper classes to locate and instantiate the specified classes in the XML file. These wrapper classes are either hand written using a set of macros, or generated from class header files using a tool. For this to be automated in Visual Studio, all the individual header files for the classes

accessed by the XML have to have a custom build step, using the command line based class wrapper generator. The output is in the form of a complete wrapper class consisting of the expected .h and .cpp file. Since the code generation is done per file and the wrapper classes need to be compiled individually, this has the potential to really slow down large builds.

The near doubling of classes will obviously increase the footprint on disk as well as in memory. Added, Autumn requires a dynamically linked library (AutumnFramework_D.dll) at runtime to parse and generate the container, which further adds to the total footprint. The need for the library makes Autumn platform specific and porting to new platforms are potentially cumbersome.

The wrapper class exposes code for DLL export, making it possible to fairly easy compile components and their wrapper classes into proper libraries for use with Autumn.

Since the XML file is parsed at runtime, and thanks to the use of wrapper classes, the configuration can be completely changed without recompilation. This design choice probably comes from an attempt to imitate the behavior of IoC containers for managed code languages such as Spring for Java and Windsor Castle for C# where reflection is present to help with class discovery and component wiring.

This comes with a huge drawback though. The XML is not statically parsed, only parsed in runtime, making it impossible to statically validate the configuration. It can only be discovered in runtime. This negates the whole benefit of the declarative model of the XML file. In contrast to most other scripting languages, XML is declarative, not procedural. This has the benefit of expressing “what” instead of “how”. By relying only on runtime parsing, the developer is back to running, and debug stepping, but with the disadvantage of not being able to read the actual code. The XML is parsed inside the external dll and will either succeed or throw an exception, making it very difficult to debug.

On page 89, *figure XXX* illustrates the sequence for interacting with the Autumn framework. The code equivalent would be that of *code example XXXI*.

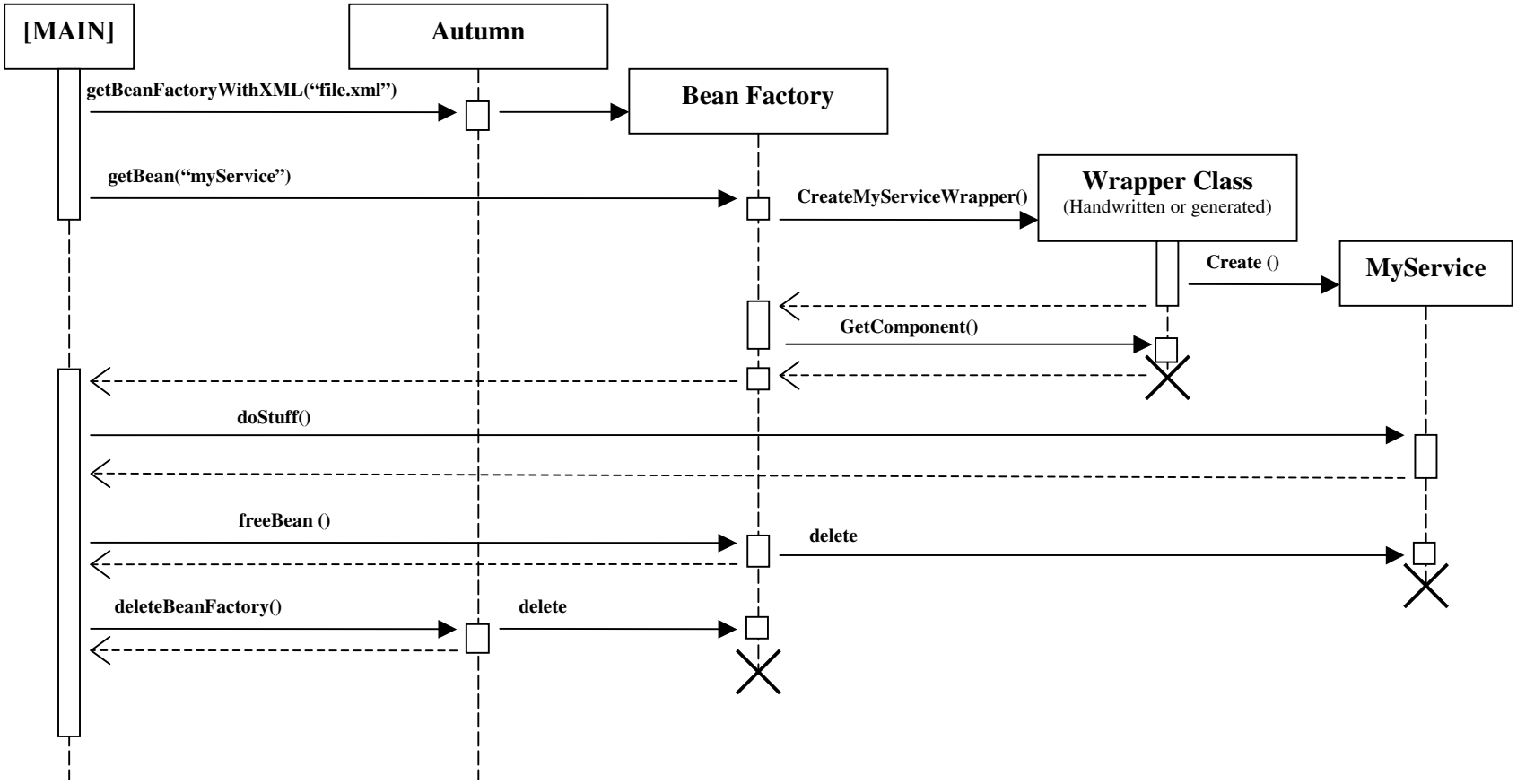


figure XXX. Autumn Framework Use Sequence

This is a simplified case. If the Component has dependencies, this will be signalled by the wrapper class having dependencies on its own to the corresponding wrapper classes. The Bean can resolve this using the XML file. When the asked for component is instantiated by its wrapper class, it has all its dependencies available and can be completely created.

It was previously stated that Autumn was not feature complete. Bellow follows a short summary of the feature set as discovered. This is an interpretation after limited use and code review and not part of the official documentation. Other features may exist, just not discovered at the time.

- Autumn configuration only supports the use of objects through pointers, or constant values of basic types. This makes it impossible to get a basic type or enum as a result from a factory and use it as argument in object instantiation. This excludes Autumn's use with many existing frameworks and programs since using enums for configuration and basic types such as integers for resource identifiers are quite common. It also excludes the use of smart pointers.
- Autumn configuration strictly deals with objects. It cannot do anything procedural such as use stand-alone functions as factory methods. Nor can it invoke static class methods, with the exception of methods that return instances of the own type, hence supporting the unfortunate Singleton pattern. The result is that it enforces a very strict object oriented model, but at the same time alienates a lot of already existing code.

To contrast the manual wiring presented in the introduction to this chapter, here follows the corresponding Autumn XML configuration code. Just as the manual wiring made use of factory methods to compose the different components, the configuration references other beans declared in the same file, just not shown here. Complete listing of the configuration file can be found in *Appendix D*.

code example XXXII. Autumn Configuration, Wiring of Game Application

```
<?xml version="1.0" encoding="utf-8"?>
<autumn xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="autumn.xsd">
  <library path="local">
    <beans>
     ...
      <bean class="Game" name="GameApp">
        <constructor-arg>
          <argument type="HGE">
            <ref bean="gfw"/>
          </argument>
          <argument type="TerrainManager">
            <ref bean="TerrainManager"/></argument>
          <argument type="ScoreComponent">
            <ref bean="ScoreComp"/></argument>
          <argument type="MenuComponent">
            <ref bean="ScoreComp"/></argument>
          <argument type="HighScoreComponent">
            <ref bean="HighScoreComp"/></argument>
          <argument type="CreditComponent">
            <ref bean="CreditComp"/></argument>
          <argument type="LevelManager">
            <ref bean="LevelManager"/></argument>
          <argument type="GameObjectManager">
            <ref bean="GameObjectManager"/></argument>
          <argument type="GameObjectFactory">
            <ref bean="GameObjectFactory"/></argument>
          <argument type="PhysicsComponent">
            <ref bean="PhysicsComp"/></argument>
          <argument type="Rect"><ref bean="CameraView"/></argument>
          <argument type="hgeFont"><ref bean="font"/></argument>
        </constructor-arg>
      </bean>
      ...
</beans>
  </library>
</autumn>
```

Going back to the evaluation criteria stated in section *4.3 IoC Container Evaluation*, the table below gives a brief summary of the result for the Autumn Framework.

table XIII. Autumn Framework Evaluation Results

Criterion	Result
Is inheritance from special classes required for the components injected, injected into?	The classes themselves do not require special inheritance, but the wrapper classes do.
Is tagging, marking of classes needed?	Wrapper classes can be created manually using a set of macros, but the actual class does not need to be tagged.

What is the overhead in terms of C++ code when retrieving and using components from the container?	The creation of the container as well as extraction of components is only a few lines of code. However, the requirement of freeing created components is an overhead that could have been avoided.
Is the structure of the XML, easy to read, easy to find errors?	The XML structure itself is simple enough, but it is very difficult to find errors since no statically analysis is done. Everything is resolved in runtime.

Did it require a custom build step? If so, was it difficult to set it up?	The individual custom build step was easy to setup. An example for Visual Studio is shown below. However, since every single component header file needs it, it quite tedious. It can be added in bulk but since most applications grow one class at a time, it can become cumbersome to set up. Also, any classes from third party frameworks or components also require wrappers, so the process can be extensive.
	Custom build step in Visual Studio 2005:
	`Command line:`
	`$(SolutionDir)Autumn\AutumnGen.exe $(InputPath) -out $(SolutionDir)`
	`Description:`
	`Generating wrapper files for $(InputName).h`
	`Outputs:`
	`$(InputName)_wrapper.h;$(InputName)_wrapper.cpp`
	`Additional Dependencies:`
	`$(SolutionDir)Autumn\AutumnGen.exe;`

Did it add or reduce flexibility in terms of switching out components?	Since all new components require wrapper classes, it is not encouraging to add new components. Further, since the XML file is parsed in runtime, it allows for great flexibility, but was quite error-prone.
Overall readability, did it add or reduce complexity?	The XML syntax is quite verbose and due to the limited set of features, it required extensive workarounds in code for otherwise getting non-conforming code.
Overall impression, did it add or reduce complexity?	Due to the error-prone system of wrapper classes and runtime parsing of XML, makes the whole framework complex and difficult to use. The lack of documentation and limitations of features makes it somewhat a burden to use.

In conclusion, Autumn Framework is not mature enough for use in commercial projects. Its relatively heavy footprint due to wrapper classes and its reliance on dynamically linked libraries makes it unsuitable if not directly unusable for embedded environments. The limited feature set in combination with the difficulties to debug was so severe that a complete configuration of Codebase II could not be made without massive refactoring. Hence, a complete XML configuration file could not be completed within the scope of the project. *Appendix D* lists an outline to a configuration file, but it is not complete.

7.8.2. PocoCapsule

PocoCapsule (version 1.0, released December 2007) is close to a full feature IoC container. It is completely non-invasive and does not require inheritance from specific classes, macro use or tagging of the production code. It thereby adheres by the behavior of a true IoC container.

It uses XML-configuration to wire classes together. The XML file is used to generate proxies in C++ code. The XML declaration has to match the POCO invocation signature for the component and can hence not be changed without recompilation. Value types such as numbers and strings are allowed to change since that does not affect the invocation the signature. Since the XML is statically validated

against its schema and generates C++ code that is compiled (static syntax and type checking), it is fairly easy to spot errors prior to runtime.

code example XXXIII. Statically Validation of XML in PocoCapsule

```
<bean id="Client" class="Client">
   <method-arg type="float" value="300.50" />
   <method-arg type="ushort" value="8080" />
</bean>
```

```
class Client
{
public:
   Client(float defaultMilliSecTimeout, unsigned short defaultPort);
};
```

An extra argument or change of order of the method arguments would result in a compile error since it would no longer match the class' signature. A replacement of 8080 with "xxx" would however not trigger a compile error since values are read from the file at runtime. It is also this feature which allows for changing the default port of the client without requiring a recompile.

In addition to the case described above, PocoCapsule also allow for complete static code generation. This option does not require the XML file at runtime and hence does not allow for change of values.

The advantage of PocoCapsule's implementation and how it uses XML is that it allows for the XML configuration to be purely declarative in contrast to procedural processes such as in code wiring or scripting languages. By making the configuration declarative it can be thought of a declaration of **what** the result should look like, not **how** it should be made. This removes any temporal coupling, declarations can be specified in any order, and thereby removes a large set of possible scenarios for programmer induced errors.

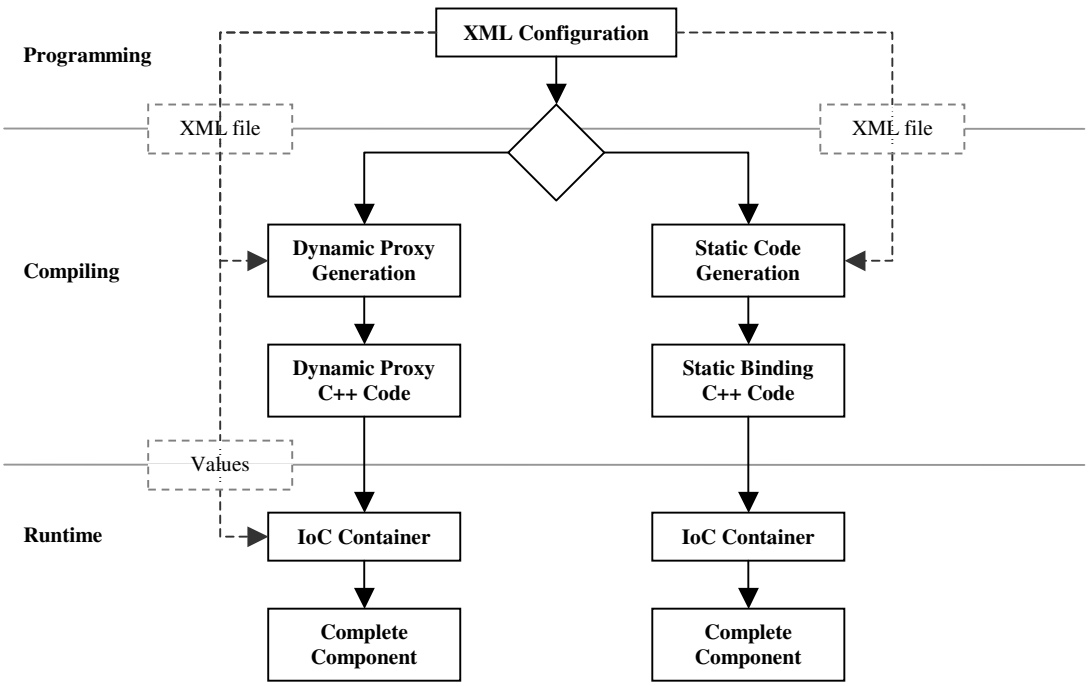


figure XXXI.PocoCapsule Use Sequence

Take the example below. In the case of the C++ code (procedural) the three statements cannot be rearranged. Creation of `A` and `B` must come before binding them together. In the XML case (declarative), the statements can come in any order and the configuration will still be valid.

code example XXXIV. Declarative vs. Procedural

```
<bean class="void" factory-method="bind">
   <method-arg ref="CompA" />
   <method-arg ref="CompB" />
</bean>
<bean id="CompA" class="A" />
<bean id="CompB" class="B" />
```

```
A* compA = new A();
B* compB = new B();
bind(compA, compB);
```

It is of course a simplified example but the shows the mechanisms at play and the superiority of a declarative model. Appendix C lists the complete XML configuration file used to wire the game application together.

Further, XML has a Meta structure understood by many other programs outside the domain of IoC. Thus can the configuration be manipulated by other programs with ease, in a context where objects are treated as data and thereby possess first class citizenship. This should not be confused with in-language functionality where most languages treat objects as data. The statement of objects as data refers to code level manipulation. Most refactoring tools don't have the semantic understanding to move around and change initialization and wiring code. The manipulation of XML structures is simple in comparison.

PocoCapsule also supports XML inclusion of other XML files, hence making it possible to break down large configurations over several files, each a small manageable unit.

code example XXXV. XML File Inclusion

```
<?xml version="1.0"?>
<!DOCTYPE poco-application-context SYSTEM
"http://www.pocomatic.com/poco-application-context.dtd"
[
<!ENTITY param-context SYSTEM "param.xml">
<!ENTITY config-context SYSTEM "config.xml">
]>
<poco-application-context>
   ...
   &param-context;
   &config-context;
   ...
</poco-application-context>
```

The XML-configuration feature of PocoCapsule is extended even further by the use of domain specific modeling (DSM). This is done by model transformation [105] using W3C Extended Stylesheet Language Transformations (XSLT) technology. This allows for high-level configuration, not requiring other than domain knowledge.

In contrast to Autumn, PocoCapsule has extensive documentation and many examples. The developers manual alone has more than one hundred pages and Ke Jin, the maintainer, has published a number of whitepapers on IoC, DI and DSM. Unfortunately, many of the examples are high level conceptual examples involving other technologies such as CORBA and other component models. Therefore, some of the finer details can be difficult to decipher. On the positive side, PocoCapsule has a small but working community and people on the discussion group are often quick to respond.

Below follows a short review of some of PocoCapsule's features as they where understood based on the experience with Codebase II, the manual and postings to the discussion group.

- In addition to generating proxies for assembly local code, PocoCapsule also support dynamic and static linking to libraries in the XML configuration.

- PocoCapsule primarily works with pointers and beans are resolved as pointers to objects allocated on the heap.
- Factory methods are expected to return by pointer. Returning by value or by reference is not explicitly supported.
- Only PocoCapsule basic types (char, uchar, short ,ushort, long, ulong, float, double, string, cstring, bean [raw pointer]) can directly be passed as arguments to invocations. Other types such as objects and structs by value, enums and function pointers need to be passed using workarounds in the form of specially constructed factory methods and macros.
- The XML configuration supports the concept of arrays as well as C++ templates.
- PocoCapsule has something referred to as an environment. The programmer can programmatically store values in the environment and use them in the configuration as a kind of default values.

code example XXXVI. PocoCapsule and Environmental Variables

```
POCO_AppEmv* env = …;
Env->setValue(“tag A”, “value A”);
Env->setValue(“max-thread”, “12”);
Ctxt = POCO_AppContext::create(“setup.xml”, “file”, env);
```

```
<bean class=”MyClass” >
   <method-arg type=”string” env-var=”tag A” />
   <method-arg type=”short” env-var=”max-thread” />
</bean>
```

- The proxy generator only generates minimal proxies based on the xml to allow for use in embedded systems and situations where memory use is an issue.
- The XML parser supports higher order transformation using XLST making it suitable for DSM.
- PocoCapsule has an extension PocoCapsule/CORBA for integration with Component-Based CORBA applications.
- PocoCapsule is neutral to component programming models and can integrate with Web Service models standards such as SCA (OASIS-SCA) or any SOAP protocol stacks or WSDL to C++ mappings.

To contrast the manual wiring presented in the introduction to this chapter, here follows the corresponding PocoCapsule XML configuration code. Just as the manual wiring made use of factory methods to compose the different components, the configuration references other beans declared in the same file, just not shown here. Complete listing of the configuration file can be found in *Appendix C*.

code example XXXVII. PocoCapsule Conf., Wiring of Game Application

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE poco-application-context SYSTEM "http://www.pocomatic.com/poco-
application-context.dtd">
<poco-application-context>
  <bean id="Game"
        class="Game"
        destroy-method="delete">
    <method-arg ref="GraphicFrameWork" />
    <method-arg ref="TerrainManager" />
    <method-arg ref="ScoreComponent" />
    <method-arg ref="MenuComponent" />
    <method-arg ref="HighScoreComponent" />
    <method-arg ref="CreditComponent" />
    <method-arg ref="LevelManager" />
    <method-arg ref="GameObjectManager" />
    <method-arg ref="GameObjectFactory" />
    <method-arg ref="PhysicsComponent" />
    <method-arg ref="CameraView" />
    <method-arg ref="TextFont" />
  </bean>
</poco-application-context>
```

Going back to the evaluation criteria stated in section *4.3 IoC Container Evaluation*, the table below gives a brief summary of the result for PocoCapsule.

table XIV. PocoCapsule Evaluation Results

Criterion	Result
Is inheritance from special classes required for the components injected, injected into?	PocoCapsule is completely non-invasive and does not require specific inheritance.
Is tagging, marking of classes needed?	PocoCapsule is completely non-invasive and does not require specific tagging or marking of classes.
What is the overhead in terms of C++ code when retrieving and using components from the container?	The creation of the container as well as extraction of components is only a few lines of code. See *code example XXXVIII PocoCapsule Container Use*.

code example XXXVIII. PocoCapsule Container Use

```
#include <pocoapp.h>
int main(int argc, char** argv)
{
   // parsing the descriptor
   POCO_AppContext* ctxt = POCO_AppContext::create("setup.xml", "file");
   // all eager singletons will be instantiated. Their
   // post-instantiation ioc methods will be invoked.
   ctxt->initSingletons();

   IService* service = static_cast<IService*>(ctxt->getBean("MyService");

   // all instantiated singletons will be destroyed
   ctxt->terminate();
   ctxt->destroy();
}
```

table XV. PocoCapsule Evaluation Results (cont.)

Criterion	Result
Is the structure of the XML, easy to read, easy to find errors?	The XML schema has only a few elements and is fairly compact. It supports nested beans which help with grouping and the id system also supports stand-alone declarations despite dependencies. Since the configuration is statically validated and the generated proxies compiled the risk of error is comparable low.

Did it require a custom build step? If so, was it difficult to set it up?	The custom build step is easy to setup. An example for Visual Studio is shown below. The custom build step is for the XML file and the setup is hence only required once per configuration file. Similar setup for complete code generation.
	The only problem is that it requires a list of all include files, making it a bit verbose. One can use a "master include" to prevent constant editing of the command line, or move the entire command to a batch file. It is easier to maintain a file rather than go into properties of the xml file, than custom build step, etc.
	Custom build step in Visual Studio 2008:
	`Command line:`
	`c:\pococapsule-cpp\bin\pxgenproxy.exe -r=gather -s=_reflx.cc -h= MstrInclFile.h $(InputFileName)`
	`Description:`
	`Performing Custom Build Step on $(InputFileName)`
	`Outputs:`
	`$(InputName)_reflx.cc`
	`Additional Dependencies:`
	`c:\pococapsule-cpp\bin\pxgenproxy.exe;`

Did it add or reduce flexibility in terms of switching out components?	It is about the same as manually written wiring, assuming the manual wiring is written correctly. The XML is easier to reorganize and move around since it is declarative rather than procedural. It has a bit of a learning curve and it is one more thing to learn compared to manual wiring. Almost anyone can read it but it can be little tricky to write, especially if the C++ code is not written with an IoC container in mind. Since it only works with pointers it may require some changes of the components interface or adapter code. Some things can be worked around directly in the xml, other requires factory-methods.
Overall readability, did it add or reduce complexity?	The configuration file made dependencies clearer, and it made it easier discover possible reuse. It forces an abstraction level that is suitable, hence enforcing a good design. If the design is bad, then it will be a hard fit to make it work with PocoCapsule.
Overall impression, did it add or reduce complexity?	If the coding convension is such that raw pointers are used, then PocoCapsule is excellent. It helps enforce a good structure and design. However, if references are used in the interfaces, or enums are used for configurations, then it can become a problem. Enums require macros or factory methods to wrap its use. Passing by reference is possible but require wrapping using factory methods.

In conclusion, PocoCapsule has all the features required of an IoC container. However, due to the complex nature of C++, with three different kinds of variable use (by value, by reference and by pointer) and PocoCapsule only supporting by pointer, it can become difficult to use with projects that extensively relies on boost smart pointers or passing objects by reference.

It has small enough footprint to fit in most environments and with the option of static code generation as alternative to dynamic proxies, it offers enough flexibility to work even on most embedded platforms. The only issue can be with how objects are allocated. The default behavior is constructing objects on the heap using new. If that is not acceptable, the new operator may have to be overridden for custom memory allocation.

PocoCapsule has reached a stage where it can be used in commercial products, but it is licensed under LGPL which may not be compatible with all business scenarios.

8. Conclusions

The opening ambition of refactoring a set of classes in the scope of a case study and in the process identify common scenarios and patterns for refactoring that brings the code to a testable state has in part fallen out well. The research questions initially stated can be answered with some confidence:

Q1 What are the impediments for code to be testable in an isolated setting?

Eight impediments have been identified:

- Asserts in Test Paths
- Global State
- Stand-Alone Functions
- Object Instantiation
- Complex Data
- Exposure of State
- Work in Constructor
- Breaking Law of Demeter

They give some guidance to what have to be eliminated in order for code to be easily tested. These impediments were found as a direct result of the case study or in referenced literature. Given more time and effort, it should be possible to break them down into more detail and use them as rules for automated code style checks.

Q2 What are the steps for refactoring code to a testable state?

Through the case study, it has been found that refactoring done to break an unwanted dependency can be said to be of either of two types:

- Deep refactoring: weakens dependencies on both Meta level and base level.
- Shallow refactoring: only weakens dependencies on base level.

The two types of refactoring reflect onto the types of patterns suggested. Two base patterns have been identified:

- Complete removal through Dependency Injection, abstracted through an interface.
- Masking dependency, by isolating the use of the dependency in a virtual method that can be overridden using subtyping and polymorphism.[30]

The steps to bring code into testable state have been outlines in a set of patterns catalogued in *Appendix A*.

The results could not formally be validated due to the small sample set that the case study involved and because of the lack of good code quality metrics for testability. Even though some metric suites have been put forth, none has come close to adequately depict codes state with regard to testability.

The experience of the case study has also resulted in a set of guidelines with regard to different aspects of unit testing, DI and IoC in an unmanaged memory setting:

- Application Partitioning

[30] Michael Feathers calls this "*Subclass and Override Method*" in [8]. It can also be done in the other direction, moving the core functionality to a base class and keeping the difficult dependencies in the original production class (now a subtype to the new base class). The base class can then be tested with ease. Feathers refers to this as "*Pushing Down Dependencies*".

- Build Times
- Dependency injection: Preferred Methods
- Application Builders, Factory Methods and IoC Containers
- Convention vs. Configuration
- Lifetime Management
- Dependency Injection and RAII
- Dependency Injection and Unmanaged Code

Q3 What are the benefits and drawbacks of the two IoC containers evaluated, with respect to the second code base, Codebase II, under investigation?

The evaluation showed that Autumn Framework was unsuitable to use due to its design of runtime parsing of the configuration file and completely dynamic wrappers generated from class header files. That made the configuration error prone and de generation of wrapper classes proved complicated and added to the complexity of the application.

PocoCapsule was found to be both easy to use and had powerful features with regard to DSM and HOT. However, the configuration sometimes required in-code workarounds if the classes interfaces had not been written with an IoC container in mind. PocoCapsule's lack of support for injection by value was also a drawback. However, the overall impression was that PocoCapsule is mature enough for many production settings.

9. Future work

Throughout the report different kind of dependencies has been described and an underlying terminology and system have at time vaguely been hinted at. Late in the project, I thought up what could be the beginning of a classification system for dependencies. Unfortunately, it was not room in the already lengthy report to pursue this further.

The classification system can briefly be described as categorizing dependencies to be of certain types.

table XVI. Dependency Types - outline

Type	Meta level	Base level	State	Implementation
I : Abstract	L	L	L	L
II : Type (stateless)	S	L	L	S
III : Instance (stateful)	S	S	S	S

S: strong coupling

L: loose coupling

Depending on something abstract such as an interface results in loose coupling on both Meta level and base level. It has loose coupling to state since abstract types do not have state. The exact implementation can also change through polymorphism and subtyping.

Depending on a concrete type creates a strong coupling on Meta level and it creates coupling to a specific implementation,. Meanwhile, it can be said to be loose on base level and since a type don't carry inherent state, different instances of that type has different state, it is considered to have loose coupling to state as well.

Depending on a specific instance will bring strong coupling on all levels:

- Meta level (the instance is of a specific type)
- Base level (that particular instance)
- The instance carry a specific state (hence coupling to state)
- The specific type carries with it a specific implementation.

These dependencies are thought to be transitive in the direction of the dependency.

It would be interesting to pursue these ideas further. Is the classification complete? Can it be formally proven that they are transitive? Can it be used to device a metric that actually say something of the health of an application? If combined with graph theory, can it be used to describe a system and its properties with regard to brittleness, extensibility, maintainability, etc?

Once validated, automated tools need to be implemented to quantify and visualize these dependencies. Such an implementation would be a suitable continuation of this project.

Another area of interest would be to formalize the patterns suggested into automated refactorings implemented for IDEs. This would require better understanding of how to get a tool to make an accurate interpretation of the semantics surrounding the problem area, but would be most useful for developers in general.

The focus of the investigation presented in this report has mainly been testability from the developer's point of view. Program performance has not been taken into account. It would therefore be interesting to look into the implications for performance in C++ programs when using DI in C++. How is program size (on disk and in memory) affected? Startup time? Will the increased use of the heap instead of traditional stack based RAII affect performance? All considerable concerns and worth looking into.

10. Bibliography and Further Reading

General publications are listed in the following format:

Author last name, first name (Year) *Title*. Publisher, printing information. ISBN

Magazine and Journal publications are listed in the following format:

Author last name, first name (Year) Title, *Journal Name*. Volume, Issue. Date. Pages. Identifer

Web resources are listed in the following format:

[site identifier] Page Title. Last name, first name (date of publication/last update)
url to resource (last date of viewing)

Podcasts (listed under Web Resources) are listed in the following format:

[site identifier] Name, #episode number. Last name, first name. (date of publication/last update)
url to resource (last date of viewing)

Information not available is left out.

10.1. Primary Sources

10.1.1. Publications

[1] Beck, Kent (1999) *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, Massachusetts. ISBN 0-321-27865-8

[2] Beck, Kent (2003) *Test-driven Development : by example*. The Addison-Wesley signature series. Addison-Wesley, Boston, Massachusetts. ISBN: 0-321-14653-0

[3] Booch, Grady. Rumbauch, James. Jacobson, Ivar (1999) *The Unified Modeling Language User Guide*. Addison-Wesley. Massachusetts, Seventh printing, 2000. ISBN: 0-201-57168-4

[4] Brooks, Fred (1975, 1995) *The Mythical Man-Month*. Addison-Wesley ISBN 0-201-00650-2 (1975 ed.), 0-201-83595-9 (1995 ed.)

[5] Chalmers, Alan F. (1999) *What is this thing called Science?* Open University Press. Third edition. ISBN 0-33-520109-1

[6] Chidamber, S. R. and Kemerer C. F. (1994) *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, vol 20, no 6, June 1994. pp. 476-493

[7] Essaiasson, Peter, et al. (2003) Metodpraktikan: konsten att studera samhälle, individ och marknad. Norstedt Juridik (2nd edition), 3rd printing, 2005, Stockholm. ISBN 913910611X

[8] Feathers, Michael (2004) *Working Effectively with Legacy Code*. Prentice Hall.

[9] Fowler, Martin (1999) *Refactoring: improving the design of existing code*. Addison Wesley Longman Inc. Westford Massachusetts, april 2006, 18th printing.

[10] Fowler, Martin (2003) *Patterns of Enterprise Application Architecture*. Pearson Education Inc. 0-321-12742-0

[11] Gamma, Erich. Helm, Richard. Johnsson, Ralph. Vlissides, John (1995) *Design Patterns Element of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-20-163361-2

[12] Goldkuhl, Göran (2008) *Artefact Science vs. Practice Science: Seeking Information System Identiy*. Linköping University.

[13] Hunt, Andrew. Thomas, David (1999) *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. ISBN 020161622X

[14] Khan, R. A. Mustafa, K. (2009) *Metric Based Testability Model for Object Oriented Design (MTMOOD)*. ACM SIGSOFT Software Engineering Notes. Volume 34, Issue 2 (March 2009). P. 1-6. ISSN:0163-5948

[15] Khurum, Mahvish (2009) *Strategic Decision Support for Software Intensive Product Management*. Blekinge Institute of Technology, Licentiate Dissertation Series No. 2009:08. ISBN: 978-91-7295-168-6

[16] Kuhn, Rich. Kacker, Raghu. Lei, Yu. Hunder, Justin (2009) Combinatorial Software Testing. *IEEE Computer Society*, 0018-9162/09, August 2009.

[17] Lieberherr, Karl .J. Holland, Ian M. (1989) Assuring Good Style for Object-Oriented Programs. *Software, IEEE*. Volume 6, Issue 5. Sep 1989. Pages 38-48. 10.1109/52.35588

[18] Martin, Robert C. (1994) *OO Design Quality Metrics. An Analysis of Dependencies*. 28 Oct, 1994.

[19] Martin, Robert C. (1996) Liskov Substitution Principle. *C++ Report*, Vol. 8, Mar 1996.

[20] Martin, Robert C. (1996) The Dependency Inversion Principle. *C++ Report*, Vol. 8, May 1996.

[21] Martin, Robert C. (1996) The Interface Segregation Principle. *C++ Report*, Vol. 8, Aug 1996.

[22] Martin, Robert C. (1996) The Open Closed Principle. *C++ Report*, Vol. 8, Jan 1996.

[23] Martin, Robert C. (2003) *Agile Software Development. Principles, Patterns, and Practices*. Pearson Education, Inc Upper Saddle River, New Jersey. ISBN 0-13-597444-5

[24] Martin, Robert C. (2008) *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. Pearson Eductaion, Inc. Stoughton, Massachusetts. ISBN 978-0-13-235088-4

[25] Meszaros, Gerard (2007) *xUnit Test Patterns, Refactoring Test Code*. Pearson Education, Inc. Westford, Massachusetts, second printing. ISBN 978-0-13-149505-0

[26] Murphy-Hill, Emerson. Black, Andrew P. (2008) *Refactoring Tools: Fitness for Purpose*. Department of Computer Science. Portland State University. Portland, Oregon. May 7, 2008

[27] Szyperski, Clemens (2002) *Component Software, Beyond Object-Oriented Programming*, second edition. Addison-Wesley. ISBN 0-201-74572-0

10.1.2. Web Resources

[28] [artima.com] Test-Driven Development. A Conversion with Martin Fowler, Part V. Venners, Bill. (2 Dec, 2002)
http://www.artima.com/intv/testdriven4.html (14 Dec 2009)

[29] [cppdepend.com] Metrics Definitions
http://www.cppdepend.com/Metrics.aspx (2 Feb 2010)

[30] [developer.com] Managed, Unmanaged, Native: What Kind of Code Is This? Gregory, Kate. (28 Apr 2003)
http://www.developer.com/net/cplus/article.php/2197621/Managed-Unmanaged-Native-What-Kind-of-Code-Is-This.htm (9 Feb 2010)

[31] [dotnetrocks.com] .NET Rocks!, #410. Uncle Bob at Oredev. Martin, Robert C. Franklin, Carl. Campbell, Richard. (8 Jan 2009)
http://perseus.franklins.net/dotnetrocks_0410_robert_martin.mp3 (13 Nov 2009)

[32] [hanselminutes.com] Hansel Minutes, #145. Solid Principles with Uncle Bob. Martin, Robert C. Hanselman, Scott. (5 Jan 2009)
http://perseus.franklins.net/hanselminutes_0145.mp3 (6 Nov 2009)

[33] [hanselminutes.com] Hansel Minutes, #146. Test Driven Development is Design- The Last Word on TDD. Bellware, Scott. Hanselman, Scott. (12 Jan 2009)
http://perseus.franklins.net/hanselminutes_0146.mp3 (14 Dec 2009)

[34] [hanselminutes.com] Hansel Minutes, #169. The Art of Unit Testing with Roy Osherove. Osherove, Roy. Hanselman, Scott. (2 Jul 2009)
http://perseus.franklins.net/hanselminutes_0169.mp3 (9 Dec 2009)

[35] [hanselminutes.com] Hansel Minutes, #171. Return of Uncle Bob. Martin, Robert C. Hanselman, Scott. (16 Jul 2009)
http://perseus.franklins.net/hanselminutes_0171.mp3 (13 Nov 2009)

[36] [InfoQ] Coplien and Martin Debate TDD, CDD and Professionalism (18 Feb 2008)
http://www.infoq.com/interviews/coplien-martin-tdd (3 Sep 2009)

[37] [ke-jin.blogspot.com] Inversion of Control Containers vs the Dependency Injection patter. Jin, Ke (11 Aug 2009)
http://ke-jin.blogspot.com/2009/08/inversion-of-control-containers-other.html (23 Sep 2009)

[38] [martinfowler.com] Inversion of Control Containers and the Dependency Injection pattern. Fowler, Martin (23 Jan 2004)
http://martinfowler.com/articles/injection.html (23 Sep 2009)

[39] [misko.hevery.com] Clean Code Talks – Dependency Injection. Hevery, Misko (Nov 11, 2008)
http://misko.hevery.com/2008/11/11/clean-code-talks-dependency-injection/ (18 Jan 2010)

[40] [misko.hevery.com] Construction Injection vs. Setter Injection. Hevery, Misko (19 Feb 2009)
http://misko.hevery.com/2009/02/19/constructor-injection-vs-setter-injection/ (23 Sep 2009)

[41] [misko.hevery.com] Design for Testability Talk. Hevery, Misko (7 Oct 2009)
http://misko.hevery.com/2009/10/07/design-for-testability-talk/ (15 Dec 2009)

[42] [misko.hevery.com] To “new” or not to “new”. Hevery, Misko (30 Sep 2009)
http://misko.hevery.com/2008/09/30/to-new-or-not-to-new/ (15 Dec 2009)

[43] [misko.hevery.com] Where Have All the New Operators gone. Hevery, Misko (10 Sep 2008)
http://misko.hevery.com/2008/09/10/where-have-all-the-new-operators-gone/ (2 Feb 2010)

[44] [misko.hevery.com] How to Think About the “new” Operator with Respect to Unit Testing. Hevery, Misko (8 Jul 2008)
http://misko.hevery.com/2008/07/08/how-to-think-about-the-new-operator/ (2 Feb 2010)

[45] [misko.hevery.com] Guide: Writing Testable Code. Wolter, Jonathan. Ruffer, Russ. Hevery, Misko
http://misko.hevery.com/attachments/Guide-Writing%20Testable%20Code.pdf (1 Feb 2010)

[46] [misko.hevery.com] Singletons are Pathological Liars. Hevery, Misko (17 Aug 2008)
http://misko.hevery.com/2008/08/17/singletons-are-pathological-liars/ (2 Feb 2010)

[47] [misko.hevery.com] Where Have All the Singletons Gone?. Hevery, Misko (21 Aug 2008)
http://misko.hevery.com/2008/08/21/where-have-all-the-singletons-gone/ (2 Feb 2010)

[48] [misko.hevery.com] Root Cause of Singletons. Hevery, Misko (25 Aug 2008)
http://misko.hevery.com/2008/08/25/root-cause-of-singletons/ (2 Feb 2010)

[49] [misko.hevery.com] Breaking the Law of Demeter is Like Looking for a Needle in the Haystack. Hevery, Misko (18 Jul 2008)

http://misko.hevery.com/2008/07/18/breaking-the-law-of-demeter-is-like-looking-for-a-needle-in-the-haystack/ (2 Feb 2010)

[50] [pbell.com] Constructor vs. Setter Injection: Constructor is Better, Bell, Peter (18 Nov 2006) http://www.pbell.com/index.cfm/2006/11/18/Constructor-vs-Setter-Injection-Constructor-is-Better (23 Sep 2009)

[51] [picocontainer.org] Inversion of Control http://www.picocontainer.org/inversion-of-control.html (11 Dec 2009)

[52] [picocontainer.org] Inversion of Control History http://www.picocontainer.org/inversion-of-control-history.html (3 Sep 2009)

[53] [pocomatic.com] Why and what of Inversion of Control. Jin, Ke. (12 Apr, 2008) http://www.pocomatic.com/docs/whitepapers/ioc/ (10 Dec, 2009)

[54] [shaun.boyblack.co.za] Constructor Injection vs Setter Injection. Smith, Shaun (1 May 2009) http://shaun.boyblack.co.za/blog/2009/05/01/constructor-injection-vs-setter-injection/ (23 Sep 2009)

[55] [springsource.com] Setter injection versus constructor injection and the use of @Required. Arendsen, Alef (11 Jul 2007) http://blog.springsource.com/2007/07/11/setter-injection-versus-constructor-injection-and-the-use-of-required/ (23 Sep 2009)

[56] [stephenwalther.com] Stephen Walther on ASP.NET MVC, TDD Tests are not Unit Tests. Walther, Stephen (11 Apr 2009) http://stephenwalther.com/blog/archive/2009/04/11/tdd-tests-are-not-unit-tests.aspx (2009-09-16)

[57] [viddler.com] Oredev 2008 – Agile – Clean Code III: Functions. Martin, Robert C. (21 Nov 2008) http://www.viddler.com/explore/oredev/videos/15/ (6 Nov 2009)

[58] [viddler.com] XP: After 10 years, why are we still talking about it? Martin, Robert C. (11 Feb 2009) http://cdn-static.viddler.com/flash/simple_publisher.swf?key=ef4eb06a (6 Nov 2009)

[59] [youtube] Google Tech Talk: OO Design for Testability. Hevery, Misko (6 Oct 2009) http://www.youtube.com/watch?v=acjvKJiOvXw (13 Nov 2009)

[60] [martinfowler.com] InversionOfControl (26 Jun 2005) http://martinfowler.com/bliki/InversionOfControl.html (3 Sep 2009)

[61] [InfoQ] The Principles of Agile Design. Bob Martin (30 Jan, 2007) http://www.infoq.com/presentations/principles-agile-oo-design (2 Nov 2009)

[62] [InfoQ] Responsive Design. Beck, Kent (4 Jun, 2009) http://www.infoq.com/presentations/responsive-design (11 Sep 2009)

[63] [Wikipedia] Codebase (7 Aug 2009) http://en.wikipedia.org/wiki/Codebase (15 Sep 2009)

[64] [code.google.com] Testability Explorer: How it Works (20 Aug 2009) http://code.google.com/p/testability-explorer/wiki/HowItWorks

[65] [Wikipedia] Test Driven Development (29 Aug 2009) http://en.wikipedia.org/wiki/Test_driven_development (3 Sep 2009)

[66] [Wikipedia] Standard Generalized Markup Language (9 Sep 2009) http://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language (10 Sep 2009)

[67] [Wikipedia] Loose coupling (15 Oct 2009) http://en.wikipedia.org/wiki/Loose_coupling (15 Dec 2009)

[68] [Wikipedia] Resource Acquisition Is Initialization (6 Nov 2009) http://en.wikipedia.org/wiki/RAII (16 Dec 2009)

[69] [Wikipedia] Agile Manifesto (8 Dec 2009) http://en.wikipedia.org/wiki/Agile_Manifesto (9 Dec 2009)

[70] [Wikipedia] Coupling (computer science) (12 Dec 2009) http://en.wikipedia.org/wiki/Coupling_(computer_science) (5 Dec 2009)

[71] [Wikipedia] Liskov Substitution Principle (13 Jan 2010) http://en.wikipedia.org/wiki/Liskov_substitution_principle (20 Jan 2010)

[72] [uiah.fi] Planning an Empirical Study. Routio, Pentti (3 August, 2007) http://www2.uiah.fi/projects/metodi/144.htm (8 dec 2009)

10.2. Secondary Sources

[73] Booch, Grady (1994) *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Co. Inc, 1994, ISBN 0-8053-5340-2

[74] Chidamber, S. R. Kemerer, C. F. (1991) Toward a metrics suite for object oriented design. *In Proc 6th ACM Conf. Object Oriented Programming. Syst., Lang. and Applicat. (OOPSLA)*, Phoenix, AZ, 1991, pp. 197-211.

[75] DeMarco, Tom (1979) *Structured Analysis and System Specification*. Yourdon Press Computing Series. Englewood Cliff, NJ.

[76] Johnson, Ralph E. Foote, Brian (1988) Designing Reusable Classes. *Journal of Object-Oriented Programming*. June/July 1988, Volume 1, Number 2, pages 22-35.

[77] Page-Jones, Meilir (1988) *The Practical Guide to Structured Systems Design*, 2d ed. Yourdon Press Computing Series. Englewood Cliff, NJ.

[78] Pressman, Roger S. Ph.D (1982) *Software Engineering - A Practitioner's Approach - Fourth Edition*. ISBN 0-07-052182-4

[79] Stevens, W. Myers, G. Constantine, L. (1974) *Structured Design*. IBM Systems Journal, 13 (2), 115-139, 1974.

[80] Stroustrup, Bjarne (1994) *The Design and Evolution of C++*. Addison-Wesley. Reading, Massachusetts. ISBN 0201543303

[81] Sweet, Richard E. (1985) *The Mesa Programming Environment*. ACM 0-89791-165-2/85/006/0216

10.3. Further Reading

10.3.1. Publications

[82] Abelson, Hal. Sussman, Jerry. Sussman, Julie (1984) *Structure and Interpretation of Computer Programs*. Cambridge: MIT Press.

[83] Meyer, Betrand (1988) *Object Oriented Software Construction*. Prentice Hall.

[84] Oshoerove, Roy (2009) *The Art of Unit Testing: With Exampls in .NET*. Manning Publications. ISBN 978-1933988276

10.3.2. Web Resources

[85] [apache.org] Apache License, Version 2.0 (Jan 2004)
http://www.apache.org/licenses/LICENSE-2.0

[86] [autumnframework] A C++ dependency injection framework like Spring in Java
http://code.google.com/p/autumnframework/

[87] [bullseye.com] BullseyeCoverage – The Authority in C++ Code Coverage
http://www.bullseye.com

[88] [C2.com] Cpp Unit Lite
http://c2.com/cgi/wiki?CppUnitLite

[89] [code.google.com] Testability Explorer: Project Home (2009)
http://code.google.com/p/testability-explorer/

[90] [cppdepend.com] CppDepend : C++ Static Anallysis Tool
http://www.cppdepend.com (11 Dec 2009)

[91] [gnu.org] GNU Lesser General Public Licence
http://www.gnu.org/copyleft/lesser.html

[92] [hanselman.com] Scott Hanselman's computerzen.com. List of .NET Dependency Injection Containers (IOC). Hanselman, Scott (13 Mar 2008)
http://www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx (11 Dec 2009)

[93] [HGE] HAAF'S Game Engine – Hardware accelerated 2D game engine
http://hge.relishgames.com/

[94] [lattix.com] Lattix Software
http://www.lattix.com/

[95] [nunit.org] NUnit – Download
http://launchpad.net/nunitv2/2.5.3/2.5.3/+download/NUnit-2.5.3.9345-src.zip (9 Feb 2010)

[96] [piccontainer.org] Dependency Injection
http://www.picocontainer.org/injection.html (11 Dec 2009)

[97] [pococapsule] An IoC and DSM framework for C/C++
http://code.google.com/p/pococapsule/

[98] [pocomatic.com] Domain Specific Modeling (DSM) in IoC Frameworks. Jin, Ke. (17 Oct 2007)
http://www.pocomatic.com/docs/whitepapers/dsm/

[99] [PRL] Programming Research Ltd.
http://www.programmingresearch.com

[100] [PRL] QA C++ | Advanced static code analysis for C++
http://www.programmingresearch.com/QACPP_MAIN.html (11 Dec 2009)

[101] [Sourceforge.net] CppUnit – C++ port of JUnit
http://sourceforge.net/projects/cppunit/

[102] [Sourceforge.net] CppUTest
http://sourceforge.net/projects/cpputest/

[103] [TypeMock.com] Typemock™ Easy unit testing.
http://site.typemock.com/

[104] [w3.org] World Wide Web Consortium
http://www.w3.org/

[105] [Wikipedia] Model transformation (22 Aug 2009)
http://en.wikipedia.org/wiki/Model_transformation

On Patterns for Refactoring Legacy C++ Code into a Testable State Using Inversion of Control

- Methodology, Implementation and Practices

Appendices

Final Thesis

Master of Science in Computer Science and Engineering
at Linköping University
by

Per Böhlin

2010

Technical Supervisor: Patrik Höglund
Enea AB
Professor: Kristian Sandahl
Department of Computer and Information Science, Linköping University

Content

Appendix A Refactoring Patterns and Scenarios

This is a collection of patterns for refactoring legacy C++ code. The term pattern might seem pretentious, but can be justified by that it consists of a series of specific steps, individual steps being a refactoring; hence resulting in patterns of refactorings refactoring patterns.

The details are optimized for use with MS Visual Studio 2005 and the VC++ compiler, but should hold valid for most advanced IDE:s and C++ compliant compilers.

The patterns all follow the same structure of a title, short general description of the problem domain followed by these three sections:

Problem: A short description of the problem the pattern is trying to solve

Procedure: A step by step instruction for refactoring.

Coding example: A coding example showing how the code changes by the steps declared in the procedure-section. `New code`; ~~`deleted code`~~.

The problem description is accompanied by a UML diagram, when relevant, showing the problem structure. A complementary diagram showing the structure after refactoring is appears after the procedure steps.

The procedure description assumes a setup similar of that in the report with a separate project for testing using cppunit.

The procedure section consists of a step-by-step instruction for safe refactoring. Among these steps are *compile* and *test*. Compile differentiate between *compile class* and *compile project*. This distinction is made because of the difference in scope that the two options offer. By just compiling the class, time can be saved as well as it offers clearer error reporting when syntax errors are limited to the class at hand.

Compile project should be understood as the entire solution if the class is used in more than one project. However, it is usually quicker to compile just a single project rather than the entire solution if it is known for sure that the class is only used in that one project.

Some compile steps are listed just for added safety, to verify the correctness of a partial change, and can hence be ignored if build time is an issue.

The step Compile and test refers to the implicit scope of the over all change. Testing in this scenario refers to any unit tests that has been put in place, other automated tests or regression tests. If no tests are in place, manual testing of the application should be done to insure proper behavior.

The coding examples are often trivial in nature but are equally valid in a more complex setting. The patterns have been discovered from working on a complex project but have simplified here to clarify the point made, save space or to eliminate the connection to proprietary work.

Error handling is done sparsely, if at all. This, not to obscure the mechanism explained. In a real world application, error handling should of course be done properly.

In *Appendix B Injection Methods*, a number of different DI-methods are shown. They will here commonly be denoted as injection paths –pathways that the injected object can take to reach inside a component and made available.

Subject class refers to the class that is being refactored; most often in an effort to bring it under test in isolation. Dependency, depend upon, or variations of such nature all depict a concrete dependency that makes it difficult to test in isolation.

Most patterns have a basic assumption of a large application where the same anti-patterns have been used repeatedly. The refactorings are designed to modify a small set of the code without breaking it; hence, making it possible to refactor a subset of the application without the requirement of commit to a change throughout the entire code base.

The patterns are mainly not strategies to forcefully create artificial seams. The deeper intension is to arrive at a state of better design with natural enabling points that can be used for other purposes than testing –such as feature extension. Therefore, some of these patterns might seam intrusive in comparison to the quick fixes offered by Michael Feathers in *Working Effectively with Legacy Code*.

Most of the refactoring patterns deal with the transformation of code to use dependency injection. Any class implementing dependency injection will face a decision on how to handle copying (either using the copy constructor or assignment operator). This problem is only dealt with in pattern A.5 Inverting Control of Object Instantiation. In the other patterns this problem is disregarded.

Since objects instantiated using dependency injection do not know what implementation they know about, they cannot do deep copies of themselves. The primary solution to prevent unintentional shallow copying is to make the copy constructor and the assignment operator private. Copying should only be allowed using special clone/copy methods where all dependencies implement such methods.

If only partial refactoring is done (leaving default constructors that instantiate dependencies), then that solution should be adopted for the copy constructor and assignment operator as well.

If shallow copying is the intention, then these measures are not necessary.

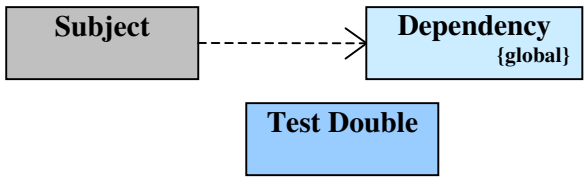
A.1. Removing Dependencies on Extern Declared Variables

In C/C++ variables and functions declared and instantiated in one module can be made visible in other modules using the `extern` declaration. This type of dependencies can sometime be difficult to detect, much less get a clear overview if it is frequently used.

Test in isolation becomes very difficult. For occasional components, the subject can be supplied with an alternative definition using the link time seem. However, this limits the injection of the test double to a test project wide scope. Several definitions are not allowed since it causes name collisions at the link stage. Hence, each test case requiring a different implementation of the depended upon class requires its unique test project. Obviously, this is not a feasible solution for a large project with multiple dependencies between components. As a consequence, this type of strong couplings must be dissolved if testing in isolation is to be achievable on a larger scale.

A.1.1. Problem

The subject that is to be tested in isolation makes use of a variable declared using the external declaration. The destructive dependency of the extern declaration is to be eliminated for this specific class, without requiring that changes propagate through out the entire application. The object is hence to have control of the instance referenced so it can be replaced with a test double.



A.1.2. Procedure

step 1) Extract interface (*A.9 Extract interface*) on the class that constitutes the dependency.

step 2) Let the class of the externally declared variable implement that interface.

step 3) In the subject class, create a private member variable of pointer type to the interface just created, with the same name as the external-declared variable.

step 4) Change the use of the dot operator to pointer-arrow operator for all usage of the variable if needed.

step 5) Assign the global variable to the newly created member variable in the constructor.

step 6) Rename the variable to follow the convention of member variables.

step 7) Compile and test. Alternatively, compile subject class and compile depended on class.

step 8) Create an injection path.

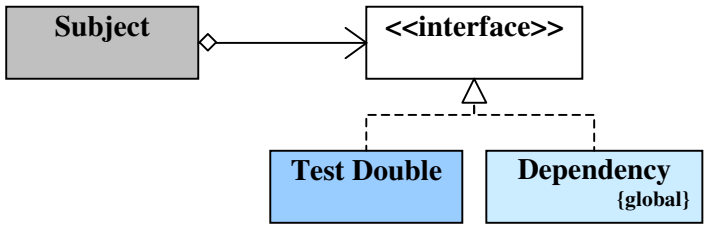
step 9) Identify all variable instantiation of the subject class and change it to match the injection path created.

step 10) Compile and test.

step 11) Remove old constructor.

step 12) Remove the external declaration from the subjects source file.

step 13) Compile and test.



A.1.3. Code example

This example operates on the assumption that a logger instance is declared in a cpp-file and used in another class, Engine, through an extern-declaration provided through an h-file.

The goal of the refactoring is to eliminate the dependency on the global variable, `g_log`, so that the logger can easily be replaced by a test double in a test situation.

global.h

```
#include ”logger.h”
extern Logger g_log;
```

global.cpp

```
#include ”global.h”
Logger g_log(); // Global instance of Logger
```

engine.h

```
class Engine
{
   Engine();
   ~Engine();
   void run();
}
```

engine.cpp

```
#include “engine.h”
#include ”global.h”
Engine::Engine(){}
Engine::~Engine(){}
void Engine::run()
{
   g_log.write(”Engine is running”);
}
```

main.cpp

```
...
int main()
{
   Engine engine = new Engine();
   Engine->run();
   delete engine;
}
```

In this situation, the Engine class cannot be tested in isolation. It depends on the concrete implementation of Logger through the global variable `g_log`.

Applying the procedure:

step 1) Extract interface (A.9 *Extract interface)* on the class that constitutes the dependency.

ilogger.h (new)

```
class ILogger
{
   ILogger(){};
   virtual ~ILogger(){};
   virtual write()=0;
}
```

step 2) Let the class of the externally declared variable implement that interface.

logger.h

```
#include "ilogger.h"
class Logger
   : public ILogger
{
   Logger();
   ~Logger();
   write();
}
```

step 3) In the subject class, create a private member variable of pointer type to the interface just created, with the same name as the external-declared variable.

engine.h

```
#include "ilogger.h"
class Engine
{
public:
   Engine();
   ~Engine();
   void run();
private:
   ILogger* g_log;   //member with same name as global variable
}
```

step 4) Change the use of the dot operator to pointer-arrow operator for all usage of the variable if needed.

step 5) Assign the global variable to the newly created member variable in the constructor.

engine.cpp

```
#include “engine.h”
#include “global.h”
...
void Engine::Engine()
{
   this->g_log = &::g_log;
}
void Engine::run()
{
   g_log.write(”Engine is running”);
   g_log->write(”Engine is running”);
}
```

step 6) Rename the variable to follow the convention of member variables.

step 7) Compile and test. Alternatively, compile subject class and compile depended on class.

engine.h

```
...
private:
   ILogger* g_log;
   ILogger* m_log;
...
```

engine.cpp

```
Engine::Engine()
{
   this->g_log = &::g_log;
   m_log = &::g_log;
}
void Engine::run()
{
   g_log->write(”Engine is running”);
   m_log->write(”Engine is running”);
}
```

step 8) Create an injection path

Use constructor, setter, interface injection depending on situation and need.

engine.h

```
...
class Engine
{
public:
   Engine();
   Engine(ILogger* logger);
...
}
```

engine.cpp (new)

```
...
Engine::Engine(ILogger* logger) : m_log(0)
{
   m_log = logger;
}
...
```

step 9) Identify all variable instantiation of the subject class and change it to match the injection path created.

If the old constructor is temporarily commented out, the compiler can help with identifying all uses of the old constructor.

step 10) Compile and test

main.cpp

```
...
#include "global.h"
int main()
{
   Engine engine = new Engine(&g_log);
   Engine->run();
   delete engine;
}
```

step 11) Remove old constructor.

step 12) Remove the external declaration from the subjects source file.

engine.h

```
...
class Engine
{
public:
   ~~Engine();~~
   Engine(ILogger* logger);
...
}
```

engine.cpp

```
~~#include "global.h"~~
...
~~Engine::Engine~~
~~{~~
~~   m_log = &::g_log;~~
~~}~~
...
```

step 13) Compile and test

Note that the primary goal is to eliminate the required inclusion of global.h –not to eliminate individual extern-declarations from the header-file. The pattern operates under the assumption that other parts of the code base still use global.h. The procedure is repeated for every global variable used by the *subject* class and externally declared in global.h until the `#include "global.h"` statement can be removed from the class definition. If global.h contains definitions needed, those can be transferred to a new file –global_definitions.h that in turn is also included in global.h. This way the project still builds while locally reducing the dependencies on global variables.

In this example, pointer-type variables have been used to enable polymorphism. Consequently, dot operator use had to be changed to pointer-arrow. If the dependency is never replaced during the lifetime of the subject and if constructor injection is used, reference type can be used instead to avoid the change of access operator.

An alternative strategy is to refactor every use of the global variable so that it can be removed from the global files. Comment out the variable from the global files and compile to see all uses of the variable. When extracting interface on the depended upon class, use *extract locally minimized interface* instead. However, this approach can be excruciating if the variable has wide use, which is usually the case since it was made global in the first place.

If the subject class is instantiated in situations that cannot make use of the new injection path, keep the old constructor until these areas can be reworked. The down side of this is of course that the meta-dependency to the global variable and its class has not been removed.

A.2. Introducing Enabling Superclass

The pattern *A.1Removing Dependencies on Extern Declared Variables* assumes repeated use on many components since the global variable often has been made global just because of its wide spread use. Consequently, many components will be getting a private member variable and possibly a set-method –depending on injection path chosen. This creates a common behavior with a large number of components that then can be extracted to a common super-class.

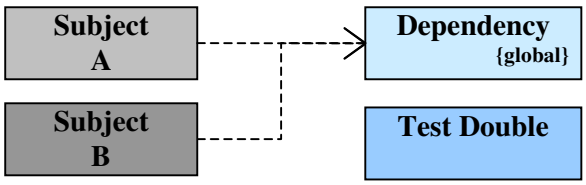
This violates the principle of keeping inheritance hierarchies simple. However, the trade off deeper inheritance structures to get the benefits of common behavior can be warranted if it adds over all clarity and structure to the application architecture.

"Enabling" in the pattern refers to that the super class introduced should provide a feature or behavior that is widely used. Logging, translation of strings, access to database is some examples of widely used components that could be suitable to bring into a class using this pattern. A superclass called Loggable could be introduced to provide logging features; its name indicating that the class now has logging features. Translatable (translation of strings) and persistent or "persistable" (access to database) are other examples of possible enabling superclasses.

This pattern could be considered a variation on interface injection.

A.2.1. Problem

A given component is used extensively through out the application through a global variable and the pattern *A.1 Removing Dependencies on Extern Declared Variables* has to be used on multiple subject classes.



A.2.2. Procedure

step 1) Extract interface (A.9 Extract interface) on the class that constitutes the dependency.

step 2) Let the class of the externally declared variable implement that interface.

step 3) Create an enabling superclass with a protected member variable of pointer type to the interface just created, with the similar name as the external-declared variable. Complete with code supporting the injection path chosen (suitable constructor or setter).

step 4) Compile the enabling superclass.

Repeat steps 5-11 on each subject class in the application that make use of the global variable

step 5) Let the subject class inherit from the new enabling superclass.

step 6) Change the use of the dot operator to pointer-arrow operator for all usage of the variable if needed. Change the variable name to match the protected member variable of the enabling superclass.

step 7) Remove the external declaration from the subjects source file.

step 8) Create an injection path (construction, setter, interface injection depending on situation and need) in the subject class to match that of the enabling superclass.

step 9) Compile subject class

step 10) Identify all variable instantiations of the subject class and change it to match the injection path created. If constructor injection is used, compiling the project will show all broken instantiation points.

step 11) Compile and test

Optional steps!

These steps eliminate the dependency on the injected class in favor of a dependency to the inherited enabling class. This move is purely a matter of taste.

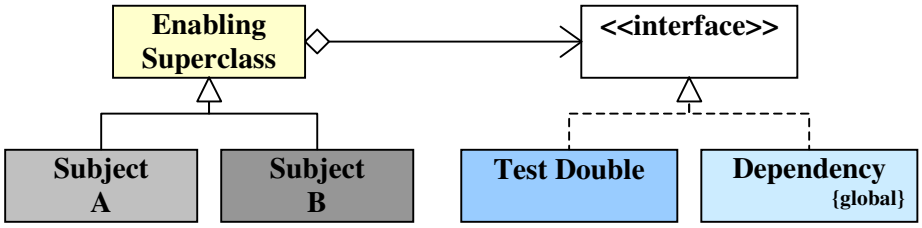
step 12) Create wrapper methods in the enabling superclass that calls the injected components methods.

step 13) Make the member variable private instead of protected.

step 14) Compile enabling superclass.

step 15) Change the calls in the subject classes from `m_variable->method(...)` to `featureMethod(...)`

step 16) Compile and test



A.2.3. Code example

In this example, the subject class `Engine` depends on the class `Language` to translate messages to a localized language. It is assumed that the `Language` class is used heavily through out the application. `Engine`, and other classes depend on a global Language variable `g_lang` and cannot be easily tested in isolation.

language.h

```
class Language
{
   Language();
   ~ Language ();
   string translate(string str);
}
```

global.h

```
#include ”language.h”
extern Language g_lang;
```

global.cpp

```
#include “global.h”
Language g_lang(); // Global instance of the Language class
```

engine.h

```
class Engine
{
   Engine();
   ~Engine();
   void run();
}
```

engine.cpp

```
#include “engine.h”
#include “global.h”
Engine::Engine(){}
Engine::~Engine(){}
Void Engine::run()
{
   ...
   string message;
   message = g_lang.translate(“SOME STANDARD STRING”);
   ...
}
```

main.cpp

```
#include “engine.h”
int main()
{
   Engine engine = new Engine};
   Engine->run();
   delete engine;
}
```

Apply the procedure:

step 1) Extract interface (A.9 *Extract interface)* on the class that constitutes the dependency.

ilanguage.h (new)

```
class ILanguage
{
   ILanguage (){};
   virtual ~ILanguage (){};
   virtual string translate(string str)=0;
}
```

step 2) Let the class of the externally declared variable implement that interface.

language.h

```
#include “ilanguage.h”
class Language: public ILanguage
{
   Language();
   ~ Language ();
   string translate(string str);
}
```

step 3) Create an enabling superclass with a protected member variable of pointer type to the interface just created, with the similar name as the external-declared variable. Complete with code supporting the injection path chosen (suitable constructor or setter).

step 4) Compile the enabling superclass

Translatable.h (new)

```
#include “ilanguage.h”
class Translatable
{
protected:
   ILanguage* m_lang;
public:
   Translatable(ILanguage* lang);
}
```

Translatable.cpp (new)

```
#include ”Translatable.h”
Translatable::Translatable (ILanguage* lang)
   : m_lang(0)
{
   m_lang = lang;
}
```

Repeat steps 5-11 on each subject class in the application

In a real world case, steps 5-11 would be repeated for all classes that use language.

step 5) Let the subject class inherit from the new enabling superclass.

engine.h

```
#include “translatable.h”
Class Engine: public Translatable
...
```

step 6) Change the use of the dot operator to pointer-arrow operator for all usage of the variable if needed. Change the variable name to match the protected member variable of the enabling superclass.

engine.cpp

```
void Engine::run()
{
   ...
   string message;
   message = g_lang.translate("SOME STANDARD STRING");
   message = m_lang->translate("SOME STANDARD STRING");
   ...
}
```

step 7) Remove the external declaration from the subjects source file.

engine.cpp

```
#include "engine.h"
#include "global.h"
```

step 8) Create an injection path (construction, setter, interface injection depending on situation and need) in the subject class to match that of the enabling superclass.

step 9) Compile subject class

engine.h

```
#include "translatable.h"
Class Engine: public Translatable
{
   Engine(ILogger* log);
   ~Engine();
void run();
}
```

engine.cpp

```
#include "engine.h"
Engine::Engine(ILanguage* lang) : Translatable(lang)
{
}
```

step 10) Identify all variable instantiations of the subject class and change it to match the injection path created.

step 11) Compile and test

main.cpp

```
#include “global.h”
#include “engine.h”
int main()
{
   Engine engine = new Engine(&g_lang);
   Engine->run();
   delete engine;
}
```

Optional steps!

These steps eliminate the dependency on the injected class in favor of a dependency to the inherited enabling class. This move is purely a matter of taste.

step 12) Create wrapper methods in the enabling superclass that calls the injected components methods.

step 13) Make the member variable private instead of protected.

step 14) Compile enabling superclass.

Translatable.h

```
#include “ilanguage.h”
Class Translatable
{
protected:
   string translate(string str);
private:
   ILanguage* m_lang;
public:
   Translatable(ILanguage* lang);
}
```

Translatable.cpp (new)

```
#include ”Translatable.h”
string Translatable::translate(string str)
{
   return m_lang->translate(str);
}
```

step 15) Change the calls in the subject classes from `m_variable->method(...)` to `featureMethod(...)`

step 16) Compile and test

engine.cpp

```
void Engine::run()
{
   ...
   string message;
   ~~message = g_lang->translate("SOME STANDARD STRING");~~
   message = translate("SOME STANDARD STRING");
   ...
}
```

This pattern does not require a complete refactoring of all classes using Language. A subset can be refactored and the build will still succeed. Refactor one component at a time and only once all dependencies on g_lang have been eliminated, remove it from the global-files.

The strength of the pattern is that changes can be made to one component at a time, while the build still lives. The down side is that the subject classes get another strong potentially destructive dependency in the form of the inheritance. Superclasses cannot be replaced with fakes during testing. However, since the enabling superclass contain very little logic and is easily tested by it self, the over all addition of complexity can in many cases be worth paying.

In this example, pointer-type variables have been used to enable polymorphism. Consequently, dot operator use had to be changed to pointer-arrow. If the dependency is never replaced during the lifetime of the subject and if constructor injection is used, reference type can be used instead to avoid the change of access operator.

In the example, the constructor of the subject class was changed to accommodate the injection path of the enabling superclass. If the constructor cannot be changed due to limiting dependencies, the same technique as in the pattern *A.1 Removing Dependencies on Extern Declared Variables* that of keeping overloaded constructors can be used. The downside is then that the meta-dependency on the global variable cannot be removed.

A.3. Remove Dependency on Stand-Alone Function

Some programs with procedural tendencies may have some key functions made available throughout the system. These functions may in turn depend on global variables, make use of expensive resources or otherwise be unsuitable for inclusion in tests. This pattern shows how strong dependencies can be removed in favor of looser ones.

The main problem with stand-alone functions (and static class methods) are that they are non-virtual. Hence, there is no object seam that can be used to override unwanted behavior. One way around this is to use function pointers as in this example.

(After this pattern was formulated, a similar pattern was found in Michael Feathers' book *Working Effectively with Legacy Code*, there called "*Replace Function with Function Pointer*". Apparently, it is a known and common solution to the problem)

A.3.1. Problem

The subject that is to be tested in isolation makes use of a function declared in a widely used header file. The function makes us of global variables or other expensive resource and should therefore not be called during testing in isolation. The object is to remove the strong dependency on the function and make it possible to replace it with a test double implementation.

A.3.2. Procedure

step 1) Add a `typedef` matching the global function's signature (optional: the `typedef` is not necessary, but it makes the code cleaner).

step 2) Add private member with the same name as the global function.

step 3) Assign the global function (use global namespace identifier ::) to the newly created variable in the constructor.

step 4) Change all calls to the global function to be based on the private member instead (not necessary if same name could be chosen and namespace scope was used properly).

step 5) Compile class.

step 6) Create an injection path (constructor, setter injection) for the new member variable.

step 7) Compile project

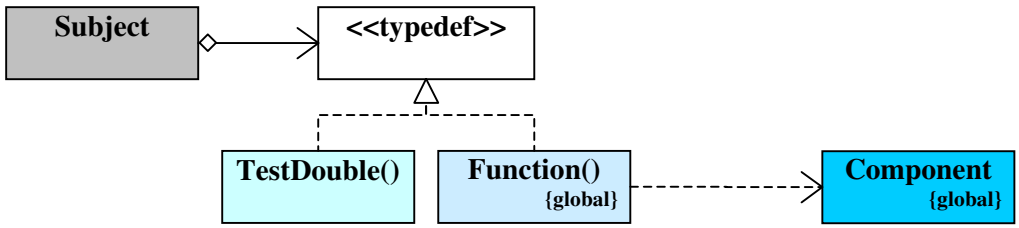
step 8) Change all instance-creation of the subject class to take the global function as argument (or test double in test case).

step 9) Compile project to make sure that it works.

step 10) Remove the old constructor.

step 11) Remove the `#include` statement for the global function declaration.

step 12) Compile and test.



A.3.3. Code example

This example operates on the assumption that a `Scenario` instance calls the widely available `displayMessageInGui()` function. The global function has a reference to a global variable that points to a user interface and displays any text sent to it. Obviously, this is not suited to be included in a unit test. Hence, the example will show how the dependency can be broken to allow for more flexibility. The example assumes that only one global function is used. Repeat the process for all global function dependencies.

The scenario class starts out like this:

scenario.h

```
class Scenario
{
public:
   Scenario();
   ~Scenario();
   void run();
};
```

scenario.cpp

```
#include "scenario.h"
#include "globalfunctions.h"
Scenario::Scenario (){}
Scenario::~Scenario (){}
void Scenario::run()
{
   displayMessageInGui("message"); //declared in globalfunctions.h
   ...
}
```

Applying the procedure:

step 1) Add a `typedef` matching the global function's signature (optional: the `typedef` is not necessary, but it makes the code cleaner).

Preferably added somewhere where it makes sense to place the definition. The `typedef` will act as our interface for the function.

Typedef for function signature (new)

```
typedef void (*DisplayFunction)(const char* msg);
```

step 2) Add private member with the same name as the global function.

scenario.h

```
class Scenario
{
public:
   Scenario();
   ~Scenario();
   void run();
private:
   DisplayFunction displayMessageInGui;
};
```

step 3) Assign the global function (use global namespace identifier ::) to the newly created variable in the constructor.

step 4) Change all calls to the global function to be based on the private member instead (not necessary if same name could be chosen and namespace scope was used properly).

scenario.cpp

```
#include "scenario.h"
#include "globalfunctions.h"
Scenario::Scenario ()
{
   This->displayMessageInGui = ::displayMessageInGui;
}
Scenario::~Scenario (){}
void Scenario::run()
{
   displayMessageInGui("message");
   ...
}
```

step 5) Compile class.

Since the interface has not changed, the newly compiled scenario should be able to link in with the rest of the object files without requiring additional recompilation. If it is possible, it can be a good idea to run an integration test to make sure that nothing has been broken.

step 6) Create an injection path (constructor, setter injection) for the new member variable.

scenario.h

```
class Scenario
{
public:
   Scenario();
   Scenario(DisplayFunction displayFunc);
   ...
};
```

scenario.cpp

```
#include “scenario.h”
#include “globalfunctions.h”
Scenario::Scenario (DisplayFunction displayFunc)
{
   displayMessageInGui = displayFunc;
}
...
```

step 7) Compile project

If the scenario class is used in only a few places, the individual files can be recompiled instead of the entire project if it saves time.

step 8) Change all instance-creation of the subject class to take the global function as argument (or test double in test case).

If constructor injection is used, the old constructor can be commented out and then the compiler will help to identify all places that require changes.

Creation of scenario instance

```
Scenario s(displayMessageInGui);
```

step 9) Compile project to make sure that it works.

step 10) Remove the old constructor.

step 11) Remove the `#include` statement for the global function declaration

scenario.h

```
class Scenario
{
public:
   Scenario();
   Scenario(DisplayFunction displayFunc);
   ~Scenario();
   void run();
private:
   DisplayFunction displayMessageInGui;
};
```

scenario.cpp

```
#include “scenario.h”
#include “globalfunctions.h”
Scenario::Scenario ()
{
   This->displayMessageInGui = ::displayMessageInGui;
}
...
```

step 12) Compile and test.

The strength of this pattern is that it removes the global dependency completely for the subject class. The down side is that it requires changes everywhere a class instance is created. If this cannot be done due to other blocking dependencies, the old constructor with the assignment of the global function to the local member variable can be kept. This however, keeps the undesirable meta-level dependency to the global variable still preventing clean reuse of the class. The benefit of this partial refactoring is that the class is at least testable.

The pattern can be applied to one class and one global function at a time, gradually refactoring the application to a better design and allowing for unit testing in isolation.

A.4. Force Strap a Component into a Test Harness: Refactoring discovery

It can sometimes be difficult to discover what refactorings need to be applied. It can then be useful to try to strap the component into a test harness and see what breaks. The hidden dependencies will show it self eventually and suitable measures can be taken to solve each dependency. This pattern shows the discovery process.

A.4.1. Problem

Legacy code is to be put under test but need refactoring. Refactoring cannot be done safely without protective tests. Usually it is dependencies to global variables, stand-alone functions or other strong coupling that prevents testing. It is not always possible to see what impediments are in place. This procedure illustrates how the compiler and linker can be used to identify the test-obstructive elements.

A.4.2. Procedure

In a stand-alone test project

step 1) Setup a TestFixture that is to hold the test code.

step 2) Add a member variable of the test subject and instantiate it in the setup method.

step 3) Add proper include paths to header files but do not link in other cpp files than the class under test.

step 4) Compile test project.

step 5) Solve link errors due to unresolved symbols:

- Global variables by using pattern: *A.1 Removing Dependencies on Extern Declared Variables*.
- Stand-alone functions by using pattern: A.3 *Remove Dependency on Stand-Alone Function*.
- Member variable types by using pattern: *A.5 Inverting Control of* Object Instantiation.
- Singletons by eliminating their use by introducing Dependency Injection (*Appendix B Injection Methods*) or by using the technique "Behavior-Modifying Subclass (Substituted Singleton)" from the book *xUnit Test Patterns* (page 586) by Gerard Meszaros..

These measures should result in loosening dependencies to concrete classes. All that is left are loose couplings to interfaces.

step 6) Write test double classes that implement the new interfaces that were created as a result of following the above mentioned refactoring patterns.

step 7) Change the instantiation code in the setup method to use test doubles to create a complete object.

step 8) Compile.

step 9) Establish an interface for the class under test.

Do this by using pattern *A.9 Extract interface* or *A.10 Extract Locally Minimized Interface (LMI)* depending on how the class is used. If the class has a lot of exposed state that is manipulated by external entities, use pattern *A.8 Extract class* to refactor the subject into a proper class. Without a properly established interface, a class will be difficult to test.

step 10) Modify the instantiation of the central class in the original application.

If this is not possible every where, the individual refactoring patterns allows for that as well. See the individual patterns to learn how to keep backward compatibility.

step 11) Compile and test in the original application.

Repeat steps 12-16 for every method/functionality or behavior that should be tested.

step 12) Write a test.

step 13) Comment out implementation code.

step 14) Run the test and watch it fail.

step 15) Uncomment the code and watch it pass.

step 16) Add another test.

A.4.3. Coding Example

This example is based around what could be a simple game. A central `Game` class makes use of a number of different components to solve different tasks:

- Creates game objects from a factory, `GameObjectFactory`, accessed through the Singleton patter.
- Writes to a log accessed through a global variable `g_log`.
- Translate messages to a local language using a stand-alone function, `translate()`.
- Makes use of a `PhysicsEngine` instantiated in the constructor.

The aim is to refactor the game class to make it testable.

main.cpp

```
#include "game.h"
int main(int argc, char* argv[])
{
   Game g;
   g.play();
   return 0;
}
```

Game.h

```
#pragma once
#include "physicsengine.h"
class Game
{
public:
   Game();
   ~Game();
   void play();
private:
   void display(const char* message);
   PhysicsEngine* m_physics;
};
```

game.cpp

```
#include "game.h"
#include "globals.h"
#include "gameobjectfactory.h"
#include "gameobject.h"
Game::Game()
{
   m_physics = new PhysicsEngine();
}
Game::~Game()
{
   delete m_physics;
}
void Game::play()
{
   g_log.write("Start playing");
   this->display(translate("START GAME"));
   GameObjectFactory* gof = GameObjectFactory::getInstance();
   GameObject* go = gof->createGameObject();
   m_physics->run(go);
   ...
}
void Game::display(const char *message)
{
   ...
}
```

globals.h

```
#pragma once
#include "logger.h"
extern Logger g_log;
const char* translate(const char* text);
```

globals.cpp

```
#include "globals.h"
Logger g_log;
const char* translate(const char* text){...}
```

physicsengine.h

```
#pragma once
#include "gameobject.h"
class PhysicsEngine
{
public:
   PhysicsEngine(void);
   ~PhysicsEngine(void);
   void run(GameObject* go);
};
```

physicsengine.cpp

```
#include "physicsengine.h"
PhysicsEngine::PhysicsEngine(void){}
PhysicsEngine::~PhysicsEngine(void){}
void PhysicsEngine::run(GameObject *go){...}
```

gameobjectfactory.h

```
#pragma once
#include "gameobject.h"
class GameObjectFactory
{
private:
   GameObjectFactory(void);
   ~GameObjectFactory(void);
   static GameObjectFactory* s_instance;
public:
   static GameObjectFactory* getInstance();
   static void destroyFactory();
   GameObject* createGameObject();
};
```

gameobjectfactory.cpp

```
#include "gameobjectfactory.h"
GameObjectFactory* GameObjectFactory::s_instance = 0;
GameObjectFactory::GameObjectFactory(void){}
GameObjectFactory::~GameObjectFactory(void){}
GameObjectFactory* GameObjectFactory::getInstance()
{
   if (s_instance == 0)
      s_instance = new GameObjectFactory();

   return s_instance;
}
void GameObjectFactory::destroyFactory()
{
   delete s_instance;
}
GameObject* GameObjectFactory::createGameObject()
{
   return new GameObject();
}
```

To start the procedure, we create a test fixture in a separate test project. In this example, we are using CppUnit as the unit-testing framework.

step 1) Setup a TestFixture that is to hold the test code.

gametest.h (new)

```
#pragma once
#include <cppunit/testfixture.h>
#include <cppunit/extensions/HelperMacros.h>
class GameTest : public CppUnit::TestFixture
{
   CPPUNIT_TEST_SUITE(GameTest);
   CPPUNIT_TEST_SUITE_END();
public:
   GameTest(void);
   ~GameTest(void);
   void setUp();
   void tearDown();
};
```

gametest.cpp (new)

```
#include "gametest.h"
CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(GameTest, "GAME");
GameTest::GameTest(void){}
GameTest::~GameTest(void){}
void GameTest::setUp()
{
}
void GameTest::tearDown()
{
}
```

step 2) Add a member variable of the test subject and instantiate it in the setup method.

gametest.h

```
#pragma once
...
class GameTest : public CppUnit::TestFixture
{
...
private:
   Game* m_game;
};
```

gametest.cpp

```
...
void GameTest::setUp()
{
   m_game = new Game();
}
void GameTest::tearDown()
{
   delete m_game;
}
```

step 3) Add proper include paths to header files but do not link in other cpp files than the class under test.

This can be achieved in many different ways. In Visual Studio, a cpp file can be added as an “existing item”. It can also be included using a regular preprocessor include directive. If the same cpp file needs to be included in more than one place it can be wrapped in custom namespaces as the example below. This is not in any way an established practice but merely a way of including class definitions in isolation.

gametest.cpp

```
/* Samething for header includes, except the “using namespace” part which
should never be used in a header file.*/
namespace GameTest_Game
{
   #include <game.cpp>
}
using namespace GameTest_Game
...
```

step 4) Compile test project.

linker output

```
1>GameTest.obj : error LNK2019: unresolved external symbol "public: __thiscall
PhysicsEngine::PhysicsEngine(void)" (??0PhysicsEngine@@QAE@XZ) referenced in
function "public: __thiscall Game::Game(void)" (??0Game@@QAE@XZ)
1>GameTest.obj : error LNK2019: unresolved external symbol "public: __thiscall
PhysicsEngine::~PhysicsEngine(void)" (??1PhysicsEngine@@QAE@XZ) referenced in
function "public: void * __thiscall PhysicsEngine::`scalar deleting
destructor'(unsigned int)" (??_GPhysicsEngine@@QAEPAXI@Z)
1>GameTest.obj : error LNK2019: unresolved external symbol "public: void
__thiscall PhysicsEngine::run(class GameObject *)"
(?run@PhysicsEngine@@QAEXPAVGameObject@@@Z) referenced in function "public:
void __thiscall Game::play(void)" (?play@Game@@QAEXXZ)
1>GameTest.obj : error LNK2019: unresolved external symbol "public: class
GameObject * __thiscall GameObjectFactory::createGameObject(void)"
(?createGameObject@GameObjectFactory@@QAEPAVGameObject@@XZ) referenced in
function "public: void __thiscall Game::play(void)" (?play@Game@@QAEXXZ)
1>GameTest.obj : error LNK2019: unresolved external symbol "public: static
class GameObjectFactory * __cdecl GameObjectFactory::getInstance(void)"
(?getInstance@GameObjectFactory@@SAPAV1@XZ) referenced in function "public:
void __thiscall Game::play(void)" (?play@Game@@QAEXXZ)
1>GameTest.obj : error LNK2019: unresolved external symbol "char const *
__cdecl translate(char const *)" (?translate@@YAPBDPBD@Z) referenced in
function "public: void __thiscall Game::play(void)" (?play@Game@@QAEXXZ)
1>GameTest.obj : error LNK2019: unresolved external symbol "public: void
__thiscall Logger::write(char const *)" (?write@Logger@@QAEXPBD@Z) referenced
in function "public: void __thiscall Game::play(void)" (?play@Game@@QAEXXZ)
1>GameTest.obj : error LNK2001: unresolved external symbol "class Logger
g_log" (?g_log@@3VLogger@@A)
```

step 5) Solve link errors due to unresolved symbols:

Not all the steps of the refactorings done are shown here, only the results. See the patterns’ descriptions for further details.

From the linker output we can see that:

A `PhysicsEngine` is used, a member variable, applying pattern *A.5 Inverting Control of* Object Instantiation to remove the dependency to the concrete class. In order to extract a proper interface from `PhysicsEngine`, the `GameObject` need to get an interface as well since it is shared between `Game` and `PhysicsEngine`.

igameobject.h (new)

```
#pragma once
class IGameObject
{
public:
   IGameObject(){};
   virtual ~IGameObject(){};
};
```

gameobject.h

```
#pragma once
#include "IGameObject.h"
class GameObject : public IGameObject
{
   ...
};
```

iphysicsengine.h (new)

```
#pragma once
#include "IGameObject.h"
class IPhysicsEngine
{
public:
   IPhysicsEngine() {};
   virtual ~IPhysicsEngine();
   virtual void run(IGameObject* go)=0;
};
```

physicsengine.h

```
#pragma once
#include "gameobject.h"
#include "igameobject.h"
#include "iphysicsengine.h"
class PhysicsEngine : public IPhysicsEngine
{
public:
   PhysicsEngine(void);
   ~PhysicsEngine(void);
   void run(GameObject* go IGameObject* go);
};
```

physicsengine.cpp

```
#include "physicsengine.h"
PhysicsEngine::PhysicsEngine(void){}
PhysicsEngine::~PhysicsEngine(void){}
void PhysicsEngine::run(GameObject *go IGameObject *go){...}
```

game.h

```
#pragma once
#include "physicsengine.h"
#include "iphysicsengine.h"
class Game
{
public:
   Game(IPhysicsEngine* physicsEngine);
   ~Game();
   void play();
   void display(const char* message);
private:
   PhysicsEngine* m_physics;
   IPhysicsEngine* m_physics;
};
```

game.cpp

```
...
Game::Game(IPhysicsEngine* physicsEngine)
{
   m_physics = new PhysicsEngine();
   m_physics = physicsEngine;
}
Game::~Game()
{
   delete m_physics;
}
...
```

GameObjectFactory, Singleton, applying *A.9 Extract interface* and then Dependency Injection from *Appendix B Constructor Injection*.

igameobjectfactory.h (new)

```
#pragma once
#include "IGameObject.h"
class IGameObjectFactory
{
public:
   IGameObjectFactory(){};
   virtual ~IGameObjectFactory(){};
   virtual IGameObject* createGameObject() = 0;
};
```

gameobjectfactory.h

```
#pragma once
#include "GameObject.h"
#include "IGameObject.h"
#include "IGameObjectFactory.h"
class GameObjectFactory : public IGameObjectFactory
{
   ...
public:
   ...
   GameObject* IGameObject* createGameObject();
};
```

gameobjectfactory.cpp

```
#include "GameObject.h"
...
GameObject* IGameObject* GameObjectFactory::createGameObject()
{
   return new GameObject();
}
```

game.h

```
#pragma once
#include "iphysicsengine.h"
#include "igameobjectfactory.h"
class Game
{
public:
   Game(IPhysicsEngine* physicsEngine, IGameObjectFactory gof);
   ~Game();
   void play();
   void display(const char* message);
private:
   IPhysicsEngine* m_physics;
   IGameObjectFactory* m_gof;
};
```

game.cpp

```
#include "Game.h"
#include "globals.h"
#include "GameObjectFactory.h"
#include "IGameObject.h"
Game::Game(IPhysicsEngine* physicsEngine, IGameObjectFactory* gof)
{
   m_physics = physicsEngine;
   m_gof = gof;
}
void Game::play()
{
   g_log.write("Start playing");
   this->display(translate("START GAME"));
   GameObjectFactory* gof = GameObjectFactory::getInstance();
   IGameObject* go = gof m_gof->createGameObject();
   m_physics->run(go);
   ...
}
...
```

The stand-alone function `translate` is dealt with applying *A.3 Remove Dependency on Stand-Alone Function*.

translationfunc.h (new)

```
typedef const char* (*translateFunc)(const char* msg);
```

game.h

```
#pragma once
#include "iphysicsengine.h"
#include "igameobjectfactory.h"
#include "translatefunc.h"
class Game
{
public:
   Game(IPhysicsEngine* physicsEngine, IGameObjectFactory gof ,
      translateFunc transfunc);
   ~Game();
   void play();
   void display(const char* message);
private:
   IPhysicsEngine* m_physics;
   IGameObjectFactory* m_gof;
   translateFunc translate;
};
```

game.cpp

```
#include "Game.h"
#include "globals.h"
#include "IGameObject.h"
Game::Game(IPhysicsEngine* physicsEngine, IGameObjectFactory* gof,
translateFunc transfunc)
{
   m_physics = physicsEngine;
   m_gof = gof;
   translate = transfunc;
}
void Game::play()
{
   g_log.write("Start playing");
   this->display(translate("START GAME")); //no need to rename translate
   IGameObject* go = m_gof->createGameObject();
   m_physics->run(go);
   ...
}
...
```

The global variable `g_log` of type `Logger` is removed using *A.1 Removing Dependencies on Extern Declared Variables*.

ilogger.h (new)

```
#pragma once
class ILogger
{
public:
   ILogger(){};
   virtual ~ILogger(){};
   virtual void write(const char* message)=0;
};
```

logger.h

```
#pragma once
#include "ILogger.h"
class Logger : public ILogger
{
   ...
};
```

game.h

```
#pragma once
#include "iphysicsengine.h"
#include "igameobjectfactory.h"
#include "translatefunc.h"
#include "ilogger.h"
class Game
{
public:
   Game(IPhysicsEngine* physicsEngine, IGameObjectFactory gof ,
      translateFunc transfunc, ILogger* log);
   ~Game();
   void play();
   void display(const char* message);
private:
   IPhysicsEngine* m_physics;
   IGameObjectFactory* m_gof;
   translateFunc translate;
   ILogger* m_log;
};
```

game.cpp

```
#include "Game.h"
#include "globals.h"
#include "IGameObject.h"
Game::Game(IPhysicsEngine* physicsEngine, IGameObjectFactory* gof,
translateFunc transfunc, ILogger* log)
{
   m_physics = physicsEngine;
   m_gof = gof;
   translate = transfunc;
   m_log = log;
}
void Game::play()
{
   g_log m_log->write("Start playing");
   this->display(translate("START GAME"));
   IGameObject* go = m_gof->createGameObject();
   m_physics->run(go);
   ...
}
...
```

`GameObject`, used by `GameObjectFactory`, `PhysicsEngine` and `Game`.GameObject was refactored during the process of removing the dependency on PhysicsEngine.

step 6) Write test double classes that implement the new interfaces that were created as a result of following the above mentioned refactoring patterns.

Test Doubles

```
class GameObjectDouble : public IGameObject
class PhysicsEngineTestDouble : public IPhysicsEngine;
class GameObjectFactoryTestDouble : public IGameObjectFactory
const char* translateTestDouble(const char* text)
class LogTestDouble : public ILogger
```

step 7) Change the instantiation code in the setup method to use test doubles to create a complete object.

TestGame.cpp

```
void GameTest::setUp()
{
   m_physics = new PhysicsEngineTestDouble();
   m_gof = new GameObjectFactoryTestDouble();
   m_log = new LogTestDouble();
   m_game  = new Game(m_physics, m_gof, translateTestDouble, m_log);
}
void GameTest::tearDown()
{
   delete m_game;
   delete m_physics;
   delete m_gof;
   delete m_log;
}
```

step 8) Compile.

step 9) Establish an interface for the class under test.

igame.h (new)

```
#pragma once
class IGame
{
public:
   IGame(){};
   virtual ~IGame(){};
   virtual void play() = 0;
};
```

game.h

```
...
#include "IGame.h"
class Game : public IGame
{
   ...
};
```

step 10)Modify the instantiation of the central class in the original application.

main.cpp

```
#include "game.h"
#include “globals.h”
#include “gameobjectfactory.h”
#include “physicsengine.h”
#include “logger.h
int main(int argc, char* argv[])
{
   Logger* log = new Logger();
   PhysicsEngine* pe = new PhysicsEngine();
   GameObjectFactory* gof = GameObjectFactory::getInstance();
   Game g(pe, gof, translate, log);
   g.play();

   delete pe;
   GameObjectFactory::destroyFactory();
   delete log;
   return 0;
}
```

step 11)Compile and test in the original application.

Steps 12-16 will not be illustrated here since they regard implement specific test cases for the class at hand and not refactoring.

Even though the process requires several refactorings, after each step (sub process) the application is still in a runnable state. The design and testability slowly improves during a process that is transparent, controlled and keeps the build alive throughout the entire process.

A.5. Inverting Control of Object Instantiation

Traditionally, dependencies are usually created in the constructor and deleted in the destructor or following the RAII practice (Resource Acquisition Is Initialization) with stack-based allocation. However, the use of new in the constructor or stack-based allocation makes testing of a class in isolation harder if not impossible.

Since the instantiated depended on object cannot easily be substituted with a fake, testing becomes very difficult. Even if the component later can be replaced by using setters, the test project still needs to know about, and create, the original dependency. If the dependency is to a class that cannot easily be created, such as a connection to a database, the test still depends on that heavy resource even if it later is replaced. Hence having resource intensive objects being created in the constructor is never an applicable option.

The changes of responsibility, moving construction and configuration of dependencies to an external entity, also have other effects. The subject class can no longer replicate it self since it does not know what implementation it has of its dependencies. A common model can be issued with a copy or clone method to allow for copying of subcomponents. In any case, the assignment operator and copy constructor should be made private in order to prevent copying through ordinary mechanisms; unless only shallow copying is expected.

This pattern should only be applied to objects where replacing the dependency is useful from a testability perspective. There is usually no need to replace a STL-vector just for the sake of testing. However, it could be useful if a collection component –linked list, vector or hash table– is interchangeable due to optimization for different configurations.

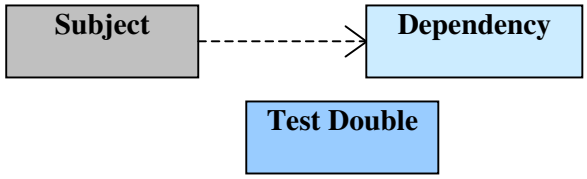
The procedure outlined uses constructor injection as injection path. The procedure can be similarly used with other injection techniques.

```
class B;
class C;

class A
{
public:
   A()
   { m_b = new B(); }
   A(const A& copy) : m_c(copy.m_c)
   { m_b = new B(*(copy.m_b)); }
   A& operator=(const A& copy)
   {
      if (this != &copy)
      {
         delete m_b;
         m_b = new B(*(copy.m_b));
         m_c = copy.m_c;
      }
      return *this;
   }
   ~A()
   {
      delete m_b;
   }
private:
   B* m_b;
   C m_c;
};
```

A.5.1. Problem

The use of the new in constructor or stack-based allocation of subcomponents prevents controlled test in isolation. The subject has a strong dependency that cannot easily be replaced with a test double.



A.5.2. Procedure

step 1) Extract an interface for the depended on component using *A.9 Extract interface* or *A.10 Extract Locally Minimized Interface (LMI)* depending on situation.

step 2) Add a constructor to the class under test that has a parameter of the interface type (pointer or reference to allow for polymorphism).

step 3) Change the type of the member variable to that of the newly created interface and assign the passed in argument in the newly added constructor to the member variable.

step 4) In the old constructor, assignment operator and copy constructor, make necessary changes to accommodate for the new type.

step 5) Comment out the original constructor and make the assignment operator and copy constructor private.

step 6) Compile the project and use syntax error list to identify all instance creation of the class and edit them to match the new constructor syntax.

step 7) If the assignment operator or copy constructor is used and a deep copy is expected, the depended on class must implement a copy/clone method to be used instead.

step 8) Compile and test.

step 9) If no other code depends on the old constructor, remove it.

A.5.3. Coding example

The purpose of the pattern is to eliminate the use of new, or direct stack-based allocation. The example will deal with a heap allocated dependency. The example further assumes that the copy constructor is used elsewhere.

Classes used in the example are a GameObject class that makes use of an AI-class. The domain use of the classes are ignored and only referred to as comments referencing “useful methods”.

gameobject.h

```
#pragma once
#include "GameObjectAI.h"
class GameObject
{
public:
   GameObject();
   GameObject(const GameObject& copy);
   ~GameObject();
   GameObject& operator=(const GameObject& copy);
   /* some usefull methods
    * ... that use some useful methods of m_ai
    */
private:
   GameObjectAI* m_ai;
};
```

gameobject.cpp

```
#include "GameObject.h"
GameObject::GameObject(void)
{
   m_ai= new GameObjectAI();
}
GameObject::GameObject(const GameObject& copy)
   : m_ai(0)
{
   if (copy.m_ai != 0)
      m_ai = new  GameObjectAI(*(copy.m_ai));
}
GameObject::~GameObject(void)
{
   delete m_ai;
}
GameObject& GameObject::operator=(const GameObject& copy)
{
   if (this != &copy)
   {
      delete m_ai;
      m_ai = 0;
      if (copy.m_ai != 0)
         m_ai = new GameObjectAI(*copy.m_ai);
   }
   return *this;
}
```

gameobjectai.h

```
#pragma once
class GameObjectAI
{
public:
   GameObjectAI(void);
   GameObjectAI(const GameObjectAI& copy);
   ~GameObjectAI(void);
   GameObjectAI& operator=(const GameObjectAI& copy);
   /* some useful methods
    * ...
    */
private:
   int m_var;
};
```

gameobjectai.cpp

```
#include "GameObjectAI.h"
GameObjectAI::GameObjectAI(void) : m_var(1){}
GameObjectAI::GameObjectAI(const GameObjectAI& copy)
{
   m_var = copy.m_var;
}
GameObjectAI::~GameObjectAI(void){}
GameObjectAI& GameObjectAI::operator=(const GameObjectAI& copy)
{
   if (this != &copy)
   {
      m_var = copy.m_var;
   }
   return *this;
}
```

step 1) Extract an interface for the depended on component using *A.9 Extract interface* or *A.10 Extract Locally Minimized Interface (LMI)* depending on situation.

This results in the addition of an interface class.

igameobjectai.h (new)

```
#pragma once
class IGameObject
{
public:
   IGameObject(){}
   virtual ~IGameObject(){};
   /* some useful virtual methods
    * ...
    */
};
```

gameobjectai.h

```
#pragma once
#include “igameobjectai.h”
class GameObjectAI : public IGameObjectAI
{
   ...
};
```

step 2) Add a constructor to the class under test that has a parameter of the interface type (pointer or reference to allow for polymorphism).

step 3) Change the type of the member variable to that of the newly created interface and assign the passed in argument in the newly added constructor to the member variable.

step 4) In the old constructor, assignment operator and copy constructor, make necessary changes to accommodate for the new type.

step 5) Comment out the original constructor and make the assignment operator and copy constructor private.

With the introduction of the interface to create the abstraction to the dependency, the copy constructor and assignment operator no longer work since they rely on knowing the exact type of the dependency. By making the copy constructor and assignment operator private, the compiler will help with preventing their use.

gameobject.h

```
#pragma once
#include "GameObjectAI.h"
class GameObject
{
public:
   //GameObject();
   GameObject(IGameObjectAI* ai);
   ~GameObject();
   /* some usefull methods
    * ... that use some useful methods of m_ai
    */
private:
   GameObject(const GameObject& copy);
   GameObject& operator=(const GameObject& copy);
private:
   ~~GameObjectAI* m_ai;~~
   IGameObjectAI* m_ai;
};
```

gameobject.cpp

```
...
/*
GameObject::GameObject(void)
{
   m_ai= new GameObjectAI();
}
*/
GameObject::GameObject(IGameObjectAI* ai)
{
   m_ai = ai;
}
GameObject::GameObject(const GameObject& copy)
{
   throw std::exception("Dependency injected classes cannot do deep copy.
      Need specific copy or clone method.");
   m_ai = new GameObjectAI(*(copy.m_ai));
}
GameObject::~GameObject(void)
{
   delete m_ai;
}
GameObject& GameObject::operator=(const GameObject& copy)
{
   throw std::exception("Dependency injected classes cannot do deep copy. Need
specific copy or clone method.");
   if (this != &copy)
   {
      delete m_ai;
      m_ai = new GameObjectAI(*copy.m_ai);
   }
   return *this;
}
...
```

step 6) Compile the project and use syntax error list to identify all instance creation of the class and edit them to match the new constructor syntax.

Not shown here.

step 7) If the assignment operator or copy constructor is used and a deep copy is expected, the depended on class must implement a copy/clone method to be used instead.

The interface class gets a clone method so that the dependency can be copied, no matter what the actual type are.

igameobjectai.h

```
#pragma once
class IGameObject
{
public:
   ...
   virtual IGameObjectAI* clone()=0;
};
```

Implementing the clone method.

gameobjectai.h

```
#pragma once
class GameObject
{
public:
   ...
   IGameObjectAI* clone();
};
```

gameobjectai.cpp

```
...
IGameObjectAI* GameObjectAI::clone()
{
   GameObjectAI* result = new GameObjectAI(*this);
   return result;
}
```

The copy constructor and assignment operator can be made public again since they are now fixed.

gameobject.h

```
#pragma once
#include "GameObjectAI.h"
class GameObject
{
public:
   //GameObject();
   GameObject(IGameObjectAI* ai);
   ~GameObject();
private:
   GameObject(const GameObject& copy);
   GameObject& operator=(const GameObject& copy);
   /* some usefull methods
    * ... that use some useful methods of m_ai
    */
private:
   IGameObjectAI* m_ai;
};
```

The copy constructor and assignment operator can now be fixed due to the existence of the dependent upon class' clone method.

gameobject.cpp

```
#include "GameObject.h"
GameObject::GameObject(void)
{
   m_ai= new GameObjectAI();
}
GameObject::GameObject(const GameObject& copy)
   : m_ai(0)
{
   throw std::exception("Dependency injected classes cannot do deep copy.
      Need specific copy or clone method.");
   if (copy.m_ai != 0)
      m_ai = copy.m_ai->clone();
}
GameObject::~GameObject(void)
{
   delete m_ai;
}
GameObject& GameObject::operator=(const GameObject& copy)
{
   throw std::exception("Dependency injected classes cannot do deep copy. Need
      specific copy or clone method.");
   if (this != &copy)
   {
      delete m_ai;
      m_ai = 0;
      if (copy.m_ai != 0)
         m_ai = copy.m_ai->clone();
   }
   return *this;
}
```

step 8) Compile and test.

Not shown here.

step 9) If no other code depends on the old constructor, remove it.

Not shown here.

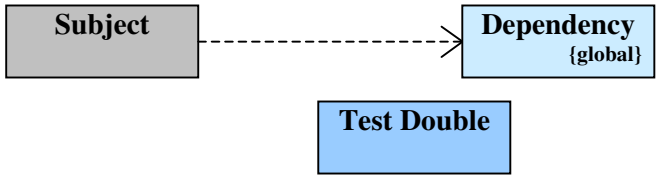
A.6. Transform Global variable to Local Using Internal Getter

This pattern solves the same problem as described in *A.1 Removing Dependencies on Extern Declared Variables*. However, this version is less intrusive and can work as a middle step if the more complete refactoring in *A.1* is too expensive to make.

(After this pattern was formulated, similar patterns was found in Michael Feathers' book *Working Effectively with Legacy Code*, there called "*Replace Global Reference with Getter*" and "*Subclass and Override Method*". Obviously it is a known and common solution to the problem)

A.6.1. Problem

The subject that is to be tested in isolation makes use of variable declared using the external declaration. The destructive dependency of the extern declaration is to be reduced for this specific class, without requiring that changes propagate through out the entire application.



A.6.2. Procedure

step 1) Extract interface (A.9 Extract interface) on the class that constitutes the dependency.

step 2) Let the class of the externally declared variable implement that interface.

step 3) Compile dependent upon class and global.cpp.

step 4) Abstract the use of the global variable in the subject class by introducing a protected virtual get-method (getter) that returns a reference or pointer of interface type to the global dependency.

step 5) Make the subject's destructor virtual if it is not already.

step 6) Compile subject class.

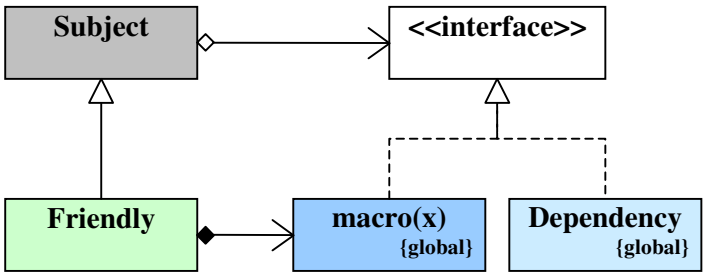
step 7) Subclass the subject and use the subclass in the test. From now on called *friendly*.

step 8) Add a member variable (reference or pointer) of compatible type to the global variable.

step 9) Add code to instantiate the member variable with a test double.

step 10) In the friendly, override the get-method so that it returns the new member variable instead of the global dependency.

step 11) Compile and test.



A.6.3. Code example

This example operates on the assumption that a logger instance is declared in *global.cpp* and used in another class, Engine, through an extern-declaration provided through *global.h*.

The goal of the refactoring is to make the dependency on the global variable, `g_log`, less definite so that the logger can be replaced with a test double in a test situation.

global.h

```
#include ”logger.h”
extern Logger g_log;
```

global.cpp

```
#include ”global.h”
Logger g_log; // Global instance of Logger
```

engine.h

```
class Engine
{
   Engine();
   ~Engine();
   void run();
};
```

engine.cpp

```
#include “engine.h”
#include ”global.h”
Engine::Engine(){}
Engine::~Engine(){}
Void Engine::run()
{
   g_log.write(”Engine is running”);
   ...
}
```

logger.h

```
class Logger
{
public:
   Logger();
   ~Logger();
   void write(const char* str);
};
```

logger.cpp

```
#include “logger.h”
Logger::Logger(){}
Logger::~Logger(){}
Void Logger::write(const char* str)
{
   ...
}
```

main.cpp

```
#include “engine.h”
...
int main()
{
   Engine* engine = new Engine();
   engine->run();
   delete engine;
}
```

In this situation the `Engine` class cannot be tested in isolation. It depends on the concrete implementation of `Logger` through the global variable `g_log`. Since the variable is global any change of state will be retained throughout the lifetime of the test environment. Any changes induced by a test will hence impact the rest of the tests that also depend on the variable. It is therefore imperative that the global state can be eliminated if isolation is to be achieved.

Applying the procedure:

step 1) Extract interface (A.9 Extract interface) on the class that constitutes the dependency.

ilogger.h (new)

```
class ILogger
{
public:
   ILogger(){};
   virtual ~Logger(){};
   virtual void write() = 0;
};
```

step 2) Let the class of the externally declared variable implement that interface.

The `Logger` class now inherits from `ILogger`, `destructor` and `write` has been made virtual.

logger.h

```
#include “ILogger.h”
class Logger : public ILogger
{
public:
   Logger();
   virtual ~Logger();
   virtual void write(const char* str);
};
```

step 3) Compile dependent upon class and global.cpp.

Due to the introduction of virtual functions, `Logger` and *global.cpp* will not be binary compatible with previous version. Even if they are recompiled, we need to do a complete recompile of any module using them before we run any tests. However, that can wait for later.

step 4) Abstract the use of the global variable in the subject class by introducing a protected virtual get-method (getter) that returns a reference or pointer of interface type to the global dependency.

step 5) Make the subject's destructor virtual if it is not already.

We encapsulated the use of `g_log` in the `getLog` method and changed any direct use of `g_log` in Engine's methods to match the new method. This affects `run` which now uses `getLog` instead of accessing `g_log` directly. Engine destructor was made virtual to allow for inheritance.

engine.h

```
#include "ilogger.h"
class Engine
{
   ...
   virtual ~Engine();
protected:
   virtual ILogger& getLog();
};
```

engine.cpp

```
...
void Engine::run()
{
   g_log.write("Engine is running");
   getLog().write("Engine is running");
}
ILogger& Engine::getLog()
{
   return (ILogger&)g_log;
}
```

step 6) Compile subject class.

Engine is now binary up to date and has also the new binary version of `Logger`.

step 7) Subclass the subject and use the subclass in the test. From now on called *friendly*.

The class `FriendlyEngine` is created as a subclass to `Engine`.

step 8) Add a member variable (reference or pointer) of compatible type to the global variable.

The member variable `log` was added to hold a `LogStub`.

step 9) Add code to instantiate the member variable with a test double.

New LogStub created in the constructor (and deleted in destructor). For even more flexibility, inject the `FriendlyEngine` with the `LogStub` using *B.1 Constructor Injection*.

step 10) In the friendly, override the get-method so that it returns the new member variable instead of the global dependency.

The virtual `getLog` method in `Engine` is overridden with a method that returns a reference to the new member variable `log`.

Friendly and Test Double (new)

```
#include "Engine.h"
#include "Logger.h"
```

```
class LogStub : public Logger
{
public:
   LogStub() : Logger() {};
   virtual ~LogStub(){};
   virtual void write(const char* str)
   {
      ...
   }
};
```

```
class FriendlyEngine : public Engine
{
public:
   TestEngine(){ log = new LogStub(); }
   virtual ~TestEngine() { delete log; }
protected:
   virtual ILogger& getLog() { return (ILogger&) *log; }
private:
   LogStub* log;
};
```

step 11) Compile and test.

The test program can now be recompiled and the tests run. A word of caution however, since the interfaces of Engine and Logger have changed; they are no longer binary compatible with previous versions. Hence any class that depend on Engine, Logger or global.cpp need to get recompiled. Some build environments may fail to detect these changes. So if the test program crashes or other peculiar behavior is noted, a complete recompile might be the solution.

The strength of this pattern is that it does not require extensive refactoring in order to work. The down side is that the global dependencies still need to be set up. It just is not used. This should be considered a shortcut. *A.1 Removing Dependencies on Extern Declared Variables* is the preferred method.

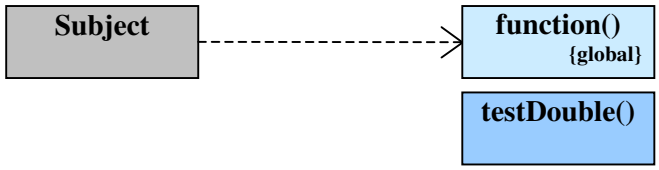
A.7. Mask Global Function Using Local Override

Some programs with procedural tendencies may have some key functions made available throughout the system. These functions may in turn depend on global variables, make use of expensive recourses or otherwise be unsuitable for inclusion in tests. This pattern shows how the function can be replaced with a specific test case implementation. This is a less invasive solution to the same problem solved in *A.3 Remove Dependency on Stand-Alone Function*.

(After this pattern was formulated, similar patterns was found in Michael Feathers' book *Working Effectively with Legacy Code*, there called "*Replace Global Reference with Getter*" and "*Subclass and Override Method*". Obviously it is a known and common solution to the problem)

A.7.1. Problem

The subject that is to be tested in isolation makes use of a function declared in a widely used header file. The function makes us of global variables or other expensive resource and should therefore not be called during testing in isolation. The object is to mask it out so a test double can be called instead



A.7.2. Procedure

step 1) Abstract the use of the global function in the subject class by introducing a protected virtual method with the same signature as the depended upon function.

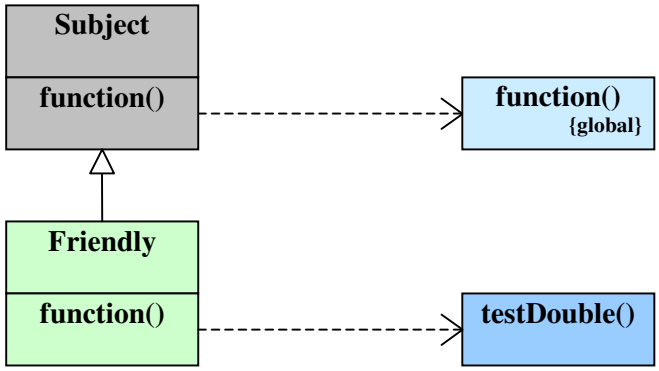
step 2) Make the subject's destructor virtual if it is not already.

step 3) Compile subject class.

step 4) Subclass the subject and use the subclass in the test. From now on called *friendly*.

step 5) In the friendly, override the abstraction-method so that it calls a safe test double or implement the test functionality directly, instead of calling the expensive global function.

step 6) Compile and test.



A.7.3. Code example

This example operates on the assumption that a `Scenario` instance calls the widely available `displayMessageInGui()` function. The global function has a reference to a global variable that points to a user interface and displays any text sent to it. Obviously this is not suited to be included in a unit test. Hence the example will show how the call can be masked away by introducing a local replacement.

The scenario class starts out like this:

scenario.h

```
class Scenario
{
public:
   Scenario();
   ~Scenario();
   void run();
};
```

scenario.cpp

```
#include “scenario.h”
#include “globalfunctions.h”
Scenario:: Scenario (){}
Scenario::~ Scenario (){}
void Scenario::run()
{
   displayMessageInGui(“message”); //declared in globalfunctions.h
   ...
}
```

Applying the procedure:

step 1) Abstract the use of the global function in the subject class by introducing a protected virtual method with the same signature as the depended upon function.

Protected virtual void method `displayMessage` is added. The name has to be different to avoid infinite recursion. Despite same names, the global function could have been accessed in the global namespace using the scope operator (::) or if the global function is in a different namespace. However, same name should be used with care since a `using namespace` could introduce the infinite recursive call that we wanted to avoid in the first place. One of the advantages of using the same name would be that we don’t have to change the call.

step 2) Make the subject’s destructor virtual if it is not already.

scenario.h

```
class Scenario
{
public:
   Scenario();
   virtual ~Scenario();
   void run();
protected:
   virtual void displayMessage(const char* message);
};
```

scenario.cpp

```
#include “scenario.h”
#include “globalfunctions.h”
Scenario:: Scenario (){}
Scenario::~ Scenario (){}
void Scenario::run()
{
   displayMessageInGui(“message”); //defined in globalfunctions.h
   displayMessage(“message”);
   ...
}
void Scenario::displayMessage (const char* message)
{
   displayMessageInGui(message);
}
```

step 3) Compile subject class.

`Scenario` is no longer binary compatible with its previous version so everything that depends on `Scenario` will eventually need to get recompiled. That can however wait for now.

step 4) Subclass the subject and use the subclass in the test. From now on called *friendly*.

step 5) In the friendly, override the abstraction-method so that it calls a safe test double or implement the test functionality directly, instead of calling the expensive global function.

Friendly and Local Test Function (new)

```
#include “scenario.h”
class FriendlyScenario : public Scenario
{
public;
   virtual ~FriendlyScenario(){};
protected:
   virtual void displayMessage(const char* message)
   {
      testFunction(message);
   }
};
testFunction(const char* message)
{
   ...
}
```

step 6) Compile and test.

The test program can now be recompiled and the tests run. A word of caution however, since the interface of Scenario and has changed; it is no longer binary compatible with previous versions. Hence any class that depend on Scenario need to be recompiled. Some build environments may fail to detect these changes. So if the test program crashes or other peculiar behavior is noted, a complete recompile might be the solution.

The strength of this pattern is that it does not require extensive refactoring in order to work. The down side is that the global dependencies still need to be set up. It just is not used. This should be considered a shortcut.

A.8. Extract class

The main purpose of this pattern is to improve the general design of a code module/application. The use of public member variable tends to leak destructive dependencies since no clear abstractions exist.

By “extracting” a proper class by moving all related code into a single class, the code becomes much easier to read, intent become clearer, code duplication can be reduced and the code is generally easier to test.

A.8.1. Problem

A struct or class has a lot of public member variables that are manipulated and used in many different places. Some code duplication exists since manipulation (calculation-use/c-use) and predicate-use (p-use) are shattered throughout the application. The code that affects the data is difficult to test since it is not concentrated to a single class.

A.8.2. Procedure

step 1) Make all member variables private.

step 2) Compile project.

step 3) Identify all compilation errors due to access violations.

step 4) In all occurrences of direct p-use of member variable, create a method and move the associated code into that method.

step 5) For all other p-uses of member variables, either move the code into a new method or create appropriate get-method.

step 6) For all c-uses, where no other variables are involved, move the entire code section into a method.

step 7) For all c-uses that involve other variables, either create appropriate methods and move the logic into those while letting the none-member variables be passed as arguments, or create getters and setters.

A.8.3. Coding Example

This coding example will only illustrate short examples of how different p- and c-use of member variables can be changed to make use of abstractions such as method calls or access through getter/setter-methods.

The given examples are a made up simplified situations and should not be seen as examples of general good programming praxis. They merely illustrate a point.

All examples makes use of the following class.

class Results

```
#include <vector>
#include <string>
class Results
{
public:
   std::vector<std::string> results;
   void* data;
};
```

step 4) In all occurrences of direct p-use of member variable, create a method and move the associated code into that method.

This example shows how direct predicate use of member variables should be moved into proper methods on the class to increase readability and reduce code duplication.

Direct P-use of member variable

```
Results r;
if (!r.results.empty())
{
   ... //do something
}
if (r.data != 0)
{
   ... //do something more
}
```

The updated version make use of encapsulation, are easier to read and the intent of the different statements have become more apparent thanks to proper naming of the introduced methods.

Direct P-use of member variable (updated)

```
#include <vector>
#include <string>
class Results
{
public:
   bool hasResults() { return (!m_results.empty()); }
   bool dataHasBeenSet() { return (m_data != 0); }
private:
   std::vector m_results;
   void* m_data;
};
```

```
Results r;
if (r.hasResults())
{
   ... //do something
}
if (r.dataHasBeenSet())
{
   ... //do somehting more
}
```

step 5) For all other p-uses of member variables, either move the code into a new method or create appropriate get-method.

This example shows indirect predicate use, where it doesn't make sense to move the entire code segment into a method, but how a get-method is employed instead.

Indirect P-use of member variable

```
Results r;
int bufferSize = 5;
if (!r.results.size >= bufferSize )
{
   ... //use buffered version
} else {
   ... //use unbuffered version
}
```

The example assumes that the bufferSize variable and its size are completely unrelated to the Result class.

Indirect P-use of member variable (updated)

```
#include <vector>
#include <string>
class Results
{
public:
   size_t getNrOfResults(){ return m_results.size(); }
private:
   std::vector<std::string> m_results;
   void* m_data;
};
```

```
Results r;
size_t bufferSize = 5;
if (r.getNrOfResults() > bufferSize)
{
   //use buffered version
}else {
   //use unbuffered version
}
```

By introduction a method as a layer of indirection, an abstraction has been created which makes the class less vulnerable to changes in the internal implementation.

step 6) For all c-uses, where no other variables are involved, move the entire code section into a method.

In this example, a print method is first accumulating all the results before printing it. Here it is assumed that the accumulation is performed elsewhere as well and code duplication can be eliminated by properly encapsulating the feature as part of the class.

Isolated C-use of member variable

```
#include <sstream>
#include <iostream>
void printResults(Results& r)
{
   std::stringstream ss;
   std::vector< std::string>::iterator it;
   for(it = r.results.begin(); it != r.results.end(); it++)
   {
      ss << (*it) << std::endl;
   }
   cout << ss.str();
}
```

The updated version make use of encapsulation, are easier to read and the intent of the different statements have become more apparent thanks to proper naming of the introduced methods.

Isolated C-use of member variable (updated)

```
#include <vector>
#include <string>
#include <sstream>
class Results
{
public:
   std::string getAccumulatedResults()
   {
      std::stringstream ss;
      std::vector<std::string>::iterator it;
      for(it = m_results.begin(); it != m_results.end(); it++)
      {
         ss << (*it) << std::endl;
      }
      return ss.str();
   }
private:
   std::vector<std::string> m_results;
   void* m_data;
};
```

```
#include <iostream>
void printResults(Results& r)
{
   std::cout << r.getAccumulatedResults();
}
```

step 7) For all c-uses that involve other variables, either create appropriate methods and move the logic into those while letting the none-member variables be passed as arguments, or create getters and setters.

This example illustrates a normally occurring required change due to making member variables private due to general c-use of the member variable.

General C-use of member variable

```
Results r;
if (somethingImportant)
{
   r.results.push_back("new result");
}
```

The exposure of the member variable makes for strong dependencies and offers no abstractions. By introducing proper methods, the class becomes easier to test since state change is regulated through its methods.

General C-use of member variable (updated)

```
#include <vector>
#include <string>
class Results
{
public:
   void addResults(const std::string& result)
   {
      m_results.push_back(result);
   }
private:
   std::vector<std::string> m_results;
   void* m_data;
};
```

```
void general()
{
   Results r;
   if (somethingImportant)
   {
      r.addResults("new result");
   }
}
```

The finalized class could look something like below. Please note that this is a fictions example and should not be read as example of well written code –in general. The example is oversimplified to illustrate the point of the kind of decisions that will need to be made when extracting a class.

class Results (updated)

```
#include <vector>
#include <string>
#include <sstream>
class Results
{
public:
   void addResults(const std::string& result)
   {
      m_results.push_back(result);
   }
   size_t getNrOfResults(){ return m_results.size(); }
   bool hasResults(){ return !m_results.empty();}
   bool dataHasBeenSet(){ return m_data != 0; }
   void setData(void* data) { m_data = data; }
   std::string getAccumulatedResults()
   {
      std::stringstream ss;
      std::vector<std::string>::iterator it;
      for(it = m_results.begin(); it != m_results.end(); it++)
      {
         ss << (*it) << std::endl;
      }
      return ss.str();
   }
private:
   std::vector<std::string> m_results;
   void* m_data;
};
```

A.9. Extract interface

Some poorly written programs have strong dependencies between classes by relying on concrete classes. The lack of abstractions tends to create rigid structures that prevents reuse as well as prevents change. This pattern illustrates a simple process for determining what methods should be included in an interface and the steps that can be taken to introduce it.

A.9.1. Problem

An application is tightly coupled due to strong dependencies in the form of direct use of concrete classes. The lack of abstractions makes the structure rigid and expensive to change.

A.9.2. Procedure

step 1) Make all methods private. If the class has public member variables apply the pattern *Extract Class* first.

step 2) Compile and thereby learn what methods are used where. If the class has many methods and there is a wide range of uses where some can be cohesively grouped together, use Extract Local Minimized Interface instead.

step 3) For each method actually used (a lot of public methods are never used and should really be private to begin with), make them public again.

step 4) Create a new abstract class with pure virtual methods matching the methods left public in the original class.

step 5) Let the original class inherit from the newly created interface.

step 6) Rewrite uses of the original class to use the interface instead.

A.9.3. Coding Example

Let assume you have a class A with a set of methods: foo, bar and baz. The class is used in various places in the application.

class A

```
class A
{
public:
   void foo();
   void bar();
   void baz();
};
```

```
{
   A a;
   a.foo();
   a.bar();
}
```

step 1) Make all methods private. If the class has public member variables apply the pattern *Extract Class* first.

class A (updated)

```
class A
{
public:
private:
   void foo();
   void bar();
   void baz();
};
```

step 2) Compile and thereby learn what methods are used where. If the class has many methods and there is a wide range of uses where some can be cohesively grouped together, use Extract Local Minimized Interface instead.

The compiler provides excellent information about in what context the class is used.

```
file.cpp(3) : error C2248: 'A::bar' : cannot access private member
declared in class 'A'
file.cpp(4) : error C2248: 'A::bar' : cannot access private member
declared in class 'A'
```

step 3) For each method actually used (a lot of public methods are never used and should really be private to begin with), make them public again.

class A (updated)

```
class A
{
public:
   void foo();
   void bar();
private:
   void baz();
};
```

step 4) Create a new abstract class with pure virtual methods matching the methods left public in the original class.

class IA (new)

```
class IA
{
public:
   virtual void foo() = 0;
   virtual void bar() = 0;
};
```

step 5) Let the original class inherit from the newly created interface.

class A (updated)

```
class A : public IA
{
   ...
};
```

step 6) Rewrite uses of the original class to use the interface instead.
Not shown here.

A.10. Extract Locally Minimized Interface (LMI)

This pattern makes use of Interface Segregation Principle (ISP). When extracting an interface from large classes that has multiple uses in different contexts, it can be beneficial to break up the interface in several smaller interfaces. Hence making use of the Interface Segregation Principle. The purpose of this refactoring is to introduce abstractions and hence make an application less rigid.

"Locally Minimized" refers to the practice of making an interface as small as possible from a specific client's or set of clients' local perspective. It is not the global context of the entire application that dictates what methods are included in the individual interfaces extracted.

A.10.1. Problem

Many different clients use a large concrete class with a lot of methods, where each client only uses a small, but cohesive, subset of the class' methods. The lack of abstractions prevents decoupling and hence the structure lacks extensibility.

A.10.2. Procedure

step 1) Make all methods private. If the class has public member variables, make use of the *Extract Class* pattern first.

step 2) Compile and thereby learn where the class is used and what methods are called. If only a single cohesive group of methods are used, employ the pattern *Extract Interface* instead.

step 3) Study the methods used and try to group them into cohesive groups where each group of methods are to form a minimized interface localized for that group of uses.

step 4) For each method actually used (a lot of public methods are never used and should really be private to begin with), make them public again.

step 5) For each group of methods identified, create an abstract class with a suitable name with pure virtual methods matching those found.

step 6) Let the original class inherit from the newly created classes and thereby implement all of the newly interfaces created.

step 7) Rewrite the uses of the original class to use the appropriate interface instead.

A.10.3. Coding Example

It is difficult to create a relevant artificial example that makes sense from a coding perspective. Therefore a very trivial example will be used to illustrate the mechanics of the pattern. In a real world application this pattern might only be applicable to very large classes that for various reasons cannot be split into smaller pieces.

class B

```
class B
{
public:
   void m1();
   void m2();
   void m3();
   void mA();
   void mB();
   void mC();
};
```

```
{
   B b;
   b.m1();
   b.m2();
}
```

```
{
   B b;
   b.mA();
   b.mB();
}
```

step 1) Make all methods private. If the class has public member variables, make use of the *Extract Class* pattern first.

class B (updated)

```
class B
{
public:
private:
   void m1();
   void m2();
   void m3();
   void mA();
   void mB();
   void mC();
};
```

step 2) Compile and thereby learn where the class is used and what methods are called. If only a single cohesive group of methods are used, employ the pattern *Extract Interface* instead.

The compiler provides excellent information about in what contexts the class is used.

```
file1.cpp(13) : error C2248: 'B::m1' : cannot access private member
declared in class 'B'
file1.cpp(14) : error C2248: 'B::m2' : cannot access private member
declared in class 'B'
file2.cpp(45) : error C2248: 'B::mA' : cannot access private member
declared in class 'B'
file2.cpp(46) : error C2248: 'B::mB' : cannot access private member
declared in class 'B'
```

step 3) Study the methods used and try to group them into cohesive groups where each group of methods are to form a minimized interface localized for that group of uses.

This can be tricky and a general recommendation cannot be provided. Many issues can be factored in. In the end, it comes down to knowledge about the application and general experience.

In this trivial example it is obvious that m1, m2 can be grouped together and mA and mB forms another cohesive unit.

step 4) For each method actually used (a lot of public methods are never used and should really be private to begin with), make them public again.

class B (updated)

```
class B
{
public:
   void m1();
   void m2();
   void mA();
   void mB();
private:
   void m3();
   void mC();
};
```

step 5) For each group of methods identified, create an abstract class with a suitable name with pure virtual methods matching those found.

class IB1 (new)

```
class IB
{
public:
   virtual void m1() = 0;
   virtual void m2() = 0;
};
```

class IB2 (new)

```
class IB
{
public:
   virtual void mA() = 0;
   virtual void mB() = 0;
};
```

step 6) Let the original class inherit from the newly created classes and thereby implement all of the newly interfaces created.

class B (updated)

```
class B : public IB1, public IB2
{
   ...
};
```

step 7) Rewrite the uses of the original class to use the appropriate interface instead.

Not shown here.

A.11. Extract Working Constructor

It might be tempting to put configuration-work in the constructor of a class. By letting the constructor create an objects dependencies as well as performing logical work to bring a class to a complete state might at first seem as a sound strategy. However, this will create strong dependencies that cannot be broken by object seams. Since constructors cannot be made virtual, once they exist, they cannot be overridden using polymorphism. Hence, effectively preventing structures that are easily extendable and testable. This pattern helps to realize and remove the dependencies created by the practice of putting work in the constructor.

A.11.1. Problem

Creation of dependencies as well as logical work creates strong dependencies that cannot be removed by object seams, hence prevent effective testing.

A.11.2. Procedure

step 1) Apply pattern Inverting Control of Object Instantiation where suitable.

step 2) Extract all work done in the constructor to a factory method and compile. Place the factory method among the class's files for easy access. Let the factory method has the same parameters as the constructor of the class. This will elucidate the class's dependencies, as all uses of member variables now will show up as compilation errors.

step 3) Fix the compilation errors by making the previous member variables factory-method local and then pass in the values of in the constructor using Dependency Injection.

step 4) Compile again. The change of the constructor's signature will cause new compilation errors elsewhere in the project. Change those places to instantiate by using the factory method instead of the constructor.

step 5) Refactor the factory method to cohesive parts.

step 6) Compile.

step 7) Apply pattern Extract class and Extract interface if warranted and change the original class's constructor parameters appropriately.

step 8) Compile and test.

step 9) If possible eliminate the factory method, else move it to the code section that create the application's components and wire them together.

step 10) Compile and update all places where instantiation of the original class is taking place.

A.11.3. Coding Example

This coding example is based on the assumption of a terrain-manager class for a graphical game. The class can render a limited set of tiles (square sprites) to illustrate the terrain within the camera view. Which tiles to render is decided by the terrain-layout. The terrain-manager has other responsibilities as well, but those will be ignored for the context of this example.

The first version of the class has a lot of work in the constructor. A texture, a terrain-layout and the layout's width and height (the number of tiles) are passed as parameters. From the texture, assumed to be a texture map, nine unique sprites are created. Based on the sprite, 21 different terrain-tiles are created. The same sprite are used several times but with different orientations.

The terrain-layout is a two-dimensional array of integers where the integer represents an index for which terrain-tile its maps against.

terrainmanager.h

```
#ifndef TERRAIN_MANAGER_H
#define TERRAIN_MANAGER_H
#include "stdafx.h"
#include "TerrainTile.h"
#include "BasicGameObject.h"
#define NR_OF_TERRAIN_SPRITE_TYPES    9
#define NR_OF_TERRAIN_TILE_TYPES      21
#define TILE_WIDTH                    60.0f
#define TILE_HEIGHT                   60.0f
class PhysicsComponent;
```

```
class TerrainManager
{
public:
   TerrainManager(HTEXTURE texture,
      int* terrainLayout=NULL, int width=0, int height=0);
   ~TerrainManager();
public:
   float getPixelWidth();
   float getPixelHeight();
   void loadTerrain(int* terrainLayout, int width, int height);
   void render(Rect cameraView);
   bool checkCollision(PhysicsComponent* physComp, BasicGameObject* bgo,
                       Rect area);
private:
   TerrainTile* getTile(int x_index, int y_index);
   int TerrainManager::getInternalIndex(int x_index, int y_index);
   int TerrainManager::getXTileIndexFromCoordinate(const Coord& coord);
   int TerrainManager::getYTileIndexFromCoordinate(const Coord& coord);
private:
   int* m_terrainLayout;
   TerrainTile* m_terrainTileTypes[NR_OF_TERRAIN_TILE_TYPES];
   hgeSprite* m_terrainTileSprites[NR_OF_TERRAIN_SPRITE_TYPES];
   HTEXTURE m_terrainTexture;
   int m_width;
   int m_height;
   float m_hotspotXOffset;
   float m_hotspotYOffset;
};
#endif
```

terrainmanager.cpp

```
#include "TerrainManager.h"
#include "PhysicsComponent.h"
```

```
TerrainManager::TerrainManager(HTEXTURE texture, int* terrainLayout, int
width, int height) :
   m_terrainTexture(texture),
   m_terrainLayout(NULL),
   m_width(width), m_height(height),
   m_hotspotXOffset(TILE_WIDTH / 2.0f),
   m_hotspotYOffset(TILE_HEIGHT / 2.0f)
{
```

```
// Create sprites from texture and store in array
   m_terrainTileSprites[0] = new hgeSprite(m_terrainTexture, 2.0f, 2.0f,
      TILE_WIDTH, TILE_HEIGHT);
   m_terrainTileSprites[1] = new hgeSprite(m_terrainTexture,
      TILE_WIDTH+5.0f, 2.0f, TILE_WIDTH, TILE_HEIGHT);
   m_terrainTileSprites[2] = new hgeSprite(m_terrainTexture,
      TILE_WIDTH*2.0f+8.0f, 2.0f, TILE_WIDTH, TILE_HEIGHT);
   m_terrainTileSprites[3] = new hgeSprite(m_terrainTexture, 2.0f,
      TILE_HEIGHT+5.0f, TILE_WIDTH, TILE_HEIGHT);
   m_terrainTileSprites[4] = new hgeSprite(m_terrainTexture,
      TILE_WIDTH+5.0f, TILE_HEIGHT+5.0f, TILE_WIDTH, TILE_HEIGHT);
   m_terrainTileSprites[5] = new hgeSprite(m_terrainTexture,
      TILE_WIDTH*2.0f+8.0f, TILE_HEIGHT+5.0f, TILE_WIDTH, TILE_HEIGHT);
   m_terrainTileSprites[6] = new hgeSprite(m_terrainTexture, 2.0f,
      TILE_WIDTH*2.0f+8.0f, TILE_WIDTH, TILE_HEIGHT);
   m_terrainTileSprites[7] = new hgeSprite(m_terrainTexture,
      TILE_WIDTH+5.0f, TILE_WIDTH*2.0f+8.0f, TILE_WIDTH, TILE_HEIGHT);
   m_terrainTileSprites[8] = new hgeSprite(m_terrainTexture,
      TILE_WIDTH*2.0f+8.0f, TILE_WIDTH*2.0f+8.0f, TILE_WIDTH,
      TILE_HEIGHT);
```

```
// Set hotspot to center of sprite;
   for (int i = 0; i < NR_OF_TERRAIN_SPRITE_TYPES; i++)
   {
      m_terrainTileSprites[i]->SetHotSpot(m_hotspotXOffset,
         m_hotspotYOffset);
      m_terrainTileSprites[i]->SetBlendMode(BLEND_COLORMUL |
         BLEND_ALPHABLEND | BLEND_NOZWRITE);
   }
```

```
    // Create terrainTiles and store in array
    m_terrainTileTypes[0] = new TerrainTile(m_terrainTileSprites[0],
       Rect(0,0,TILE_WIDTH,TILE_HEIGHT));
    m_terrainTileTypes[1] = new TerrainTile(m_terrainTileSprites[6],
       Rect(0,0,TILE_WIDTH,TILE_HEIGHT));
    m_terrainTileTypes[2] = new TerrainTile(m_terrainTileSprites[3],
       Rect(0,0,TILE_WIDTH,TILE_HEIGHT/2.0f), M_PI_2);
    m_terrainTileTypes[3] = new TerrainTile(m_terrainTileSprites[6],
       Rect(0,0,TILE_WIDTH,TILE_HEIGHT), M_PI_2);
    m_terrainTileTypes[4] = new TerrainTile(m_terrainTileSprites[3],
       Rect(0,0,TILE_WIDTH/2.0f,TILE_HEIGHT));
    m_terrainTileTypes[5] = new TerrainTile(m_terrainTileSprites[8],
       Rect(0,0,0,0));
    m_terrainTileTypes[6] = new TerrainTile(m_terrainTileSprites[3],
       Rect(TILE_WIDTH/2.0f,0,TILE_WIDTH,TILE_HEIGHT), M_PI);
    m_terrainTileTypes[7] = new TerrainTile(m_terrainTileSprites[6],
       Rect(0,0,TILE_WIDTH,TILE_HEIGHT), -M_PI_2);
    m_terrainTileTypes[8] = new TerrainTile(m_terrainTileSprites[3],
       Rect(0,TILE_HEIGHT/2.0f,TILE_WIDTH,TILE_HEIGHT), -M_PI_2);
    m_terrainTileTypes[9] = new TerrainTile(m_terrainTileSprites[6],
       Rect(0,0,TILE_WIDTH,TILE_HEIGHT), M_PI);
    m_terrainTileTypes[10] = new TerrainTile(m_terrainTileSprites[1],
       Rect(0,0,TILE_WIDTH,TILE_HEIGHT));
    m_terrainTileTypes[11] = new TerrainTile(m_terrainTileSprites[7],
       Rect(TILE_WIDTH/2.0f,TILE_HEIGHT/2.0f,TILE_WIDTH,TILE_HEIGHT),
          M_PI);
    m_terrainTileTypes[12] = new TerrainTile(m_terrainTileSprites[7],
       Rect(0,TILE_HEIGHT/2.0f,TILE_WIDTH/2.0f,TILE_HEIGHT), -M_PI_2);
    m_terrainTileTypes[13] = new TerrainTile(m_terrainTileSprites[7],
       Rect(TILE_WIDTH/2.0f,0,TILE_WIDTH,TILE_HEIGHT/2.0f), M_PI_2);
    m_terrainTileTypes[14] = new TerrainTile(m_terrainTileSprites[7],
       Rect(0,0,TILE_WIDTH/2.0f,TILE_HEIGHT/2.0f));
    m_terrainTileTypes[15] = new TerrainTile(m_terrainTileSprites[5],
       Rect(9,2,TILE_WIDTH-7,TILE_HEIGHT-2));
    m_terrainTileTypes[16] = new TerrainTile(m_terrainTileSprites[4],
       Rect(0,0,TILE_WIDTH/2.0f,TILE_HEIGHT));
    m_terrainTileTypes[17] = new TerrainTile(m_terrainTileSprites[4],
       Rect(TILE_WIDTH/2.0f,0,TILE_WIDTH,TILE_HEIGHT), M_PI);
    m_terrainTileTypes[18] = new TerrainTile(m_terrainTileSprites[4],
       Rect(0,0,TILE_WIDTH,TILE_HEIGHT/2.0f), M_PI_2);
    m_terrainTileTypes[19] = new TerrainTile(m_terrainTileSprites[4],
       Rect(0,TILE_HEIGHT/2.0f,TILE_WIDTH,TILE_HEIGHT), -M_PI_2);
    m_terrainTileTypes[20] = new TerrainTile(m_terrainTileSprites[2],
       Rect(0,0,TILE_WIDTH,TILE_HEIGHT));

    loadTerrain(terrainLayout, width, height);
}
```

```
TerrainManager::~TerrainManager(void)
{
   if (m_terrainLayout != NULL) {
      delete [] m_terrainLayout;
      m_terrainLayout = NULL;
   }
   if (m_terrainTexture != NULL) {
      for(int i = 0; i < NR_OF_TERRAIN_TILE_TYPES; i++) {
         delete m_terrainTileTypes[i];
      }
      for(int i = 0; i < NR_OF_TERRAIN_SPRITE_TYPES; i++) {
         delete m_terrainTileSprites[i];
      }
   }
}
... //some defintions are not shown here
```

step 1) Apply pattern Inverting Control of Object Instantiation where suitable.
In this example we are not applying this step.

step 2) Extract all work done in the constructor to a factory method and compile. Place the factory method among the class's files for easy access. Let the factory method has the same parameters as the constructor of the class. This will elucidate the class's dependencies, as all uses of member variables now will show up as compilation errors.

By placing the factory method in the Terrain managers code files, it becomes easy for compilation segments that make use of the terrain manager to find the declaration and definition of the new method.

terrainmanager.h (updated)

```
...
TerrainManager* TerrainManagerFactory(HTEXTURE texture,
   int* terrainLayout = NULL, int width = 0, int height = 0);
```

terrainmanager.cpp (updated)

```
...
TerrainManager* TerrainManagerFactory(HTEXTURE texture,
   int* terrainLayout, int width, int height)
{
// Create sprites from texture and store in array
   m_terrainTileSprites[0] = new hgeSprite(m_terrainTexture, 2.0f, 2.0f,
      TILE_WIDTH, TILE_HEIGHT);
   ...
   return new TerrainManager(texture, terrainLayout, width,
      height);
}
```

step 3) Fix the compilation errors by making the previous member variables factory-method local and then pass in the values of in the constructor using Dependency Injection.

The constructor has been updated to take the external dependencies as arguments.

terrainmanager.h (updated)

```
...
   ~~TerrainManager(HTEXTURE texture,~~
      ~~int* terrainLayout=NULL, int width=0, int height=0);~~
TerrainManager(TerrainTile*terrainTileTypes[NR_OF_TERRAIN_TILE_TYPES],
     hgeSprite* terrainTileSprites[NR_OF_TERRAIN_SPRITE_TYPES],
     float hotSpotXOffset, float hotSpotYOffset,
     int* terrainLayout = NULL, int width = 0, int height = 0);
```

The factory method has been changed to do the work and pass in the result in the constructor of the Terrain Manager class.

terrainmanager.cpp (updated)

```
...
TerrainManager* TerrainManagerFactory(HTEXTURE texture, int*
terrainLayout, int width, int height)
{
   TerrainTile* terrainTileTypes[NR_OF_TERRAIN_TILE_TYPES];
   hgeSprite* terrainTileSprites[NR_OF_TERRAIN_SPRITE_TYPES];
   // Create sprites from texture and store in array
   terrainTileSprites[0] = new hgeSprite(texture, 2.0f, 2.0f,
      TILE_WIDTH, TILE_HEIGHT);
   terrainTileSprites[1] = new hgeSprite(texture, TILE_WIDTH+5.0f,
      2.0f, TILE_WIDTH, TILE_HEIGHT);
   terrainTileSprites[2] = new hgeSprite(texture, TILE_WIDTH*2.0f+8.0f,
      2.0f, TILE_WIDTH, TILE_HEIGHT);
   terrainTileSprites[3] = new hgeSprite(texture, 2.0f,
      TILE_HEIGHT+5.0f, TILE_WIDTH, TILE_HEIGHT);
   terrainTileSprites[4] = new hgeSprite(texture, TILE_WIDTH+5.0f,
      TILE_HEIGHT+5.0f, TILE_WIDTH, TILE_HEIGHT);
   terrainTileSprites[5] = new hgeSprite(texture, TILE_WIDTH*2.0f+8.0f,
      TILE_HEIGHT+5.0f, TILE_WIDTH, TILE_HEIGHT);
   terrainTileSprites[6] = new hgeSprite(texture, 2.0f,
      TILE_WIDTH*2.0f+8.0f, TILE_WIDTH, TILE_HEIGHT);
   terrainTileSprites[7] = new hgeSprite(texture, TILE_WIDTH+5.0f,
      TILE_WIDTH*2.0f+8.0f, TILE_WIDTH, TILE_HEIGHT);
   terrainTileSprites[8] = new hgeSprite(texture, TILE_WIDTH*2.0f+8.0f,
      TILE_WIDTH*2.0f+8.0f, TILE_WIDTH, TILE_HEIGHT);
```

```
// Set hotspot to center of sprite;
   float hotSpotXOffset = (TILE_WIDTH / 2.0f);
   float hotSpotYOffset = (TILE_HEIGHT / 2.0f);
   for (int i = 0; i < NR_OF_TERRAIN_SPRITE_TYPES; i++)
   {
      terrainTileSprites[i]->SetHotSpot(hotSpotXOffset, hotSpotYOffset);
      terrainTileSprites[i]->SetBlendMode(BLEND_COLORMUL | BLEND_ALPHABLEND |
         BLEND_NOZWRITE);
   }
```

```
   // Create terrainTiles and store in array
   terrainTileTypes[0] = new TerrainTile(terrainTileSprites[0],
      Rect(0,0,TILE_WIDTH,TILE_HEIGHT));
   terrainTileTypes[1] = new TerrainTile(terrainTileSprites[6],
      Rect(0,0,TILE_WIDTH,TILE_HEIGHT));
   ... [2] ... [18] ...
   terrainTileTypes[19] = new TerrainTile(terrainTileSprites[4],
      Rect(0,TILE_HEIGHT/2.0f,TILE_WIDTH,TILE_HEIGHT), -M_PI_2);
   terrainTileTypes[20] = new TerrainTile(terrainTileSprites[2],
      Rect(0,0,TILE_WIDTH,TILE_HEIGHT));

   return new TerrainManager(terrainTileTypes, terrainTileSprites,
hotSpotXOffset, hotSpotYOffset, terrainLayout, width, height);}
```

```
TerrainManager::TerrainManager(
      TerrainTile* terrainTileTypes[NR_OF_TERRAIN_TILE_TYPES],
      hgeSprite* terrainTileSprites[NR_OF_TERRAIN_SPRITE_TYPES],
      float hotSpotXOffset, float hotSpotYOffset,
      int* terrainLayout, int width, int height
   ) :
   m_hotspotXOffset(hotSpotXOffset),
   m_hotspotYOffset(hotSpotYOffset),
   m_terrainLayout(NULL),
   m_width(width), m_height(height)
{
   memcpy(m_terrainTileTypes, terrainTileTypes
           sizeof(TerrainTile*)*NR_OF_TERRAIN_TILE_TYPES);
   memcpy(m_terrainTileSprites, terrainTileSprites,
           sizeof(hgeSprite*)*NR_OF_TERRAIN_SPRITE_TYPES);
   loadTerrain(terrainLayout, width, height);
}
```

step 4) Compile again. The change of the constructor's signature will cause new compilation errors elsewhere in the project. Change those places to instantiate by using the factory method instead of the constructor.

Not shown here.

step 5) Refactor the factory method to cohesive parts.

Break up the factory methods into speciallisted function that expresses intent and makes the structure clearer. An example is not shown here.

step 6) Compile.

Verify that the changes made have not broken any code syntactically.

step 7) Apply pattern Extract class and Extract interface if warranted and change the original class's constructor parameters appropriately.

The independent steps are not shown here. The result could look something like the example below.

terrainmanager.h (updated)

```
... //includes not shown
class TerrainManager : public ITerrainManager
{
public:
   TerrainManager(ITerrainTileTypes* terrainTileTypes);
   virtual ~TerrainManager();
   CoordType getPixelWidth();
   CoordType getPixelHeight();
   void loadTerrain(int* terrainLayout, int width, int height);
   void render(Rect cameraView);
   bool checkCollision(IPhysicsComponent* physComp, BasicGameObject* bgo,
           Rect area);
private:
   TerrainTile* getTile(int x_index, int y_index);
   int getInternalIndex(int x_index, int y_index);
   int getXTileIndexFromCoordinate(const Coord& coord);
   int getYTileIndexFromCoordinate(const Coord& coord);
private:
   int* m_terrainLayout;
   int m_layoutWidth;
   int m_layoutHeight;
   ITerrainTileTypes& m_terrainTileTypes;
};
```

terrainmanager.cpp (updated)

```
...
TerrainManager::TerrainManager(ITerrainTileTypes* terrainTileTypes)
     :
        m_terrainLayout(0),
        m_layoutWidth(0), m_layoutHeight(0),
        m_terrainTileTypes(*terrainTileTypes)
{
}
```

```
TerrainManager::~TerrainManager()
{
   delete[] m_terrainLayout;
}
```

ITerrainTileTypes.h (new)

```
#pragma once
#include "terraintile.h" //not included in example
class ITerrainTileTypes
{
public:
   virtual ~ITerrainTileTypes(){};
   virtual TerrainTile*& operator[](const int i) = 0;
   virtual float GetTileWidth() = 0;
   virtual float GetTileHeight() = 0;
};
```

The rest of the classes (`TerrainTileTypes, ITileSprites, TileSprites, ITerrainManager`) are not shown here.

step 8) Compile and test.

Not shown here.

step 9) If possible eliminate the factory method, else move it to the code section that create the application's components and wire them together.

Not shown here.

step 10) Compile and update all places where

Not shown here.

By moving the work in the constructor to a factory method, the dependencies were easily revealed. By breaking down the factory method into smaller cohesive parts, it became easier to divide the responsibilities into separate classes, hence abiding to the Single Responsibility Principle.

A.12. Move Assert to Where it Really Matters

Traditional asserts, when triggered, halts the entire program. It is obviously not acceptable for single test to halt the entire test run. Custom asserts that offer better integration and communication with the test framework is therefore needed.

Also, asserts placed in constructors, and to some extent also setters, prevents the use of null pointers to be passed in during construction of objects under test. This can pose a real hindrance to producing simple clean tests. By placing asserts where they are actually needed, testing can become easier. The downside is of course that you allow incomplete objects to exist.

A.12.1. Problem

The use of asserts in constructors prevent efficient testing with null instead of real objects where such are not needed. Also, asserts as a way of enforcing a contract prevents the verification of said contract through testing.

A.12.2. Procedure

For contracts:

step 1) Use custom asserts that throw exceptions to verify breach of contract.

For null-checks:

step 2) Move the checks to the methods where the dependency is actually used.

A.12.3. Coding Example

step 1) Use custom asserts that throw exceptions to verify breach of contract.

By using the custom asserts as the example below, the boundaries of a contract enforced by asserts can be verified without breaking the entire test suite.

Custom ASSERT definition

```
#ifdef _TEST
#include "assertexception.h"
#define ASSERT(_expression) ((!(_expression)) ? \
   throw AssertException(#_expression, __FILE__, __LINE__) \
      : ((void)0))
#else
#include <cassert>
#define ASSERT(_expression) assert(_expression)
#endif
```

assertexception.h

```
#pragma once
#include <exception>
#include <string>
class AssertException :
   public std::exception
{
public:
   AssertException(const char* expression, const char* file, int line);
   virtual ~AssertException();
   virtual const char* what() const;
private:
   std::string _msg;
};
```

assertexception.cpp

```
#include "AssertException.h"
#include <sstream>
AssertException::AssertException(const char* expression,
   const char* file, int line)
{
   std::ostringstream s;
   s << file << "(" << line << ")";
   s << " Assertion failed: " << expression;
   _msg = s.str();
}

AssertException::~AssertException(){}

const char* AssertException::what() const
{
   return _msg.c_str();
}
```

step 2) Move the checks to the methods where the dependency is actually used.

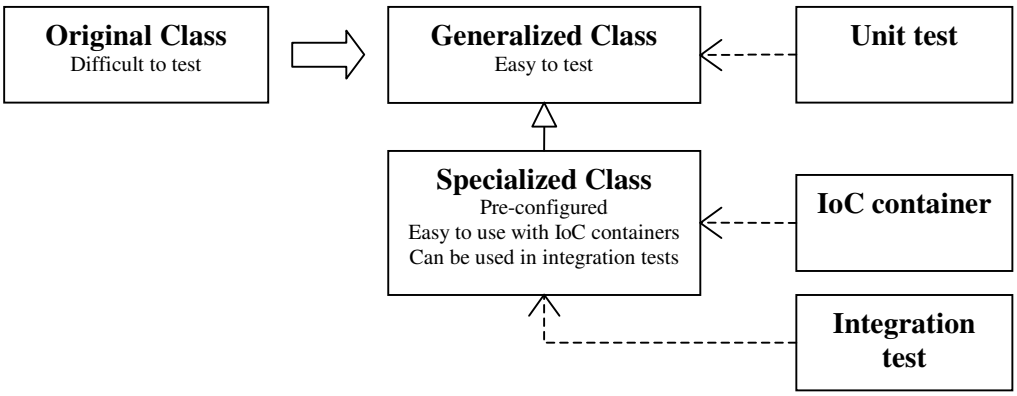
By moving asserts from the constructor to the methods where dependencies are actually used. This way some test cases can be simplified by allowing for null-pointers instead of always being forced to use real object when passing dependencies during construction of the instance under test.

No code example is provided for this part

A.13. Simplified Configuration Through Specialization

This pattern is more about separating concerns to balance the need for high-level conceptual configuration using an IoC container and that of testability for individual classes.

The method used is to refactor the original class to a state where it is generalized to allow for multiple configuration options and then restore the original configuration by subclassing from the generalized class.



A.13.1. Problem

A possible scenario is that of a class that has been allowed to do work in the constructor with the justification of strictly sticking to encapsulation and thereby simplifying application configuration. Application configuration should be kept at a high level of abstraction. Too many details risk obscuring the big picture. This pattern shows how the separation of concerns, that of implementation and configuration has been split into two classes, achieving both high level abstractions for IoC configuration as well as reasonable testing.

A.13.2. Procedure

step 1) Generalize the class by applying pattern: Extract Working Constructor and Inverting Control of Object Instantiation as needed.

step 2) Subclass the generalized class as appropriate and provide default configurations.

A.13.3. Coding Example

This example shows how the generalized class hgeFont, a font class in the HGE graphic framework can be subclassed to reduce the amount of detailed configuration needed.

Specialization Through Subclassing

```
class MenuItemFont : public hgeFont
{
public:
   MenuItemFont(const char* fontFilePath) : hgeFont(fontFilePath)
   {
      this->SetScale(0.8f);
      this->SetColor(ARGB(0xA0,0xAF,0xAF,0xA0));
   }
};
```

```
class SelectedMenuItemFont : public hgeFont
{
public:
   SelectedMenuItemFont (const char* fontFilePath) : hgeFont(fontFilePath)
   {
      this->SetScale(0.8f);
      this->SetColor(ARGB(0xEE,145,21,23));
   }
};
```

```
Class MenuItemShadowFont : public hgeFont
{
public:
   MenuItemShadowFont(const char* fontFilePath) : hgeFont(fontFilePath)
   {
      this->SetScale(1.0);
      this->SetColor(ARGB(0xA0,0x10,10,10));
   }
};
```

Appendix B Injection Methods

This is a summary with code examples of the different injection methods. Unlike the pattern, scenario description this only consists of a simple code example and a brief explanation.

B.1. Constructor Injection

Using constructor injection, any dependencies are injected as arguments to constructor with suitable parameters. In the example that follows, the class `PersistentPersonalManagerWithAudit` uses/depends on a database and a logger. `PersistentPersonalManagerWithAudit` holds pointers to interfaces that the classes `SQLDatabase` and `AuditLogger` implements. As seen, instead of `PersistentPersonalManagerWithAudit` creating its own dependencies, they are handed to it as arguments to the constructor. In a test scenario, it is trivial to use a *fake* or *friendly* implementation of database and logger instead of using the production classes. Logger writing to file or a data retrieved by accessing a database over network will be significant slower and require more setup. Further, it is also quite easy to replace the SQL-database implementation with a more lightweight file based version if it would turn out to be appropriate in a new production setting.

Constructor Injection

```
class PersistentPersonalManagerWithAudit
{
public:
   PersistentPersonalManagerWithAudit (IDatabase* db, ILogger* log)
      : m_db(0), m_log(0)
   {
      m_db = db;
      m_log = log;
   }
private:
   IDatabase* m_db;
   ILogger* m_log;
};
```

```
{
   IDatabase* db = new SQLDatabase();
   ILogger* log = new AuditLogger();
   PersistentPersonalManagerWithAudit* ppmwa
      = new PersistentPersonalManagerWithAudit(db, log);
   ... //do some work
   delete ppmwa;
   delete log;
   delete db;
}
```

B.2. Setter Injection

Setter injection relies on the use of so called setter methods. It is simply a method that is used to give a value, reference or pointer to a member variable. In the example that follows, the class `PersistentPersonalManagerWithAudit` on a database and a logger. `PersistentPersonal-ManagerWithAudit` holds pointers to interfaces that the classes `Database` and `Logger` implements. As seen, instead of `PersistentPersonalManagerWithAudit` creating its own dependencies, they are passed to it using the set-methods.

Setter Injection

```
class PersistentPersonalManagerWithAudit
{
public:
   PersistentPersonalManagerWithAudit () : m_db(0), m_log(0){}
   setDatabase(IDatabase* db) {m_db = db;}
   setLogger(ILogger* log) {m_log = log;}
private:
   IDatabase* m_db;
   ILogger* m_log;
};
```

```
{
   IDatabase* db = new SQLDatabase();
   ILogger* log = new AuditLogger();
   PersistentPersonalManagerWithAudit* ppmwa
      = new PersistentPersonalManagerWithAudit();
   ppmwa->setDatabase(db);
   ppmwa->setLogger(log);
   ... //do some work
   delete ppmwa;
   delete log;
   delete db;
}
```

B.3. Interface Injection

Interface injection uses, as the name suggests, an interface as the injection mechanism. An interface is defined that allows for the injection of the depended upon component. The dependent component implements said interface. Interface Injection is very similar to setter injection in terms of the lingual mechanism, but thanks to the interface, a framework or other entity that knows about the interface can handle the injection.

Interface Injection

```
class PersistenceAspect
{
public:
   virtual void injectDatabase(IDatabase* db) = 0;
};
```

```
class LoggingAspect
{
public:
   virtual void injectLogger(ILogger* log) = 0;
};
```

```
class PersistentPersonalManagerWithAudit
   : public PersistanceAspect, public LoggingAspect
{
public:
   void injectDatabase(IDatabase* db){m_db = db;}
   void injectLogger(ILogger* log){m_log = log;}
private:
   IDatabase* m_db;
   ILogger* m_log;
};
```

```
class AspectInjectionFactory
{
public:
   AspectInjectionFactory()
   {
      m_db = new SQLDatabase();
      m_log = new AuditLogger();
   }
   ~InjectionFactory()
   {
      delete m_db;
      delete m_log;
   }
   void injectPersistenceAspect(PersistenceAspect* pa)
   {
      pa->injectDatabase(m_db);
   }
   void injectLoggingAspect(LoggingAspect* la)
   {
      la->injectLogger(m_log);
   }
private:
   IDatabase* m_db;
   ILogger* m_log;
};
```

```
{
   PersistentPersonalManagerWithAudit* ppmwa
      = new PersistentPersonalManagerWithAudit();
   AspectInjectionFactory f;
   f.injectPersistenceAspect(ppmwa);
   f.injectLoggingAspect(ppmwa);
   ... //do some work
   delete ppmwa;
}
```

B.4. Template Injection

Template injection makes use of the C++ template engine as a facilitator for the injection of interchangeable component types. Below follows a brief code example showing its use.

Template Injection

```
template<class DB, class LOG>
class PersistentPersonalManagerWithAudit
{
public:
   PersistentPersonalManagerWithAudit()
private:
   DB m_db;
   LOG m_log;
};
```

```
#include “sqldatabase.h”
#include “auditlogger.h”
PersistentPersonalManagerWithAudit<SQLDatabase, AuditLogger> ppmwa;
... //do some work
```

As seen, with template injection, the extra layer of interfaces for the depended upon components can be neglected. However, the components cannot be replaced at runtime.

Appendix C Pococapsule Configuration XML

This configuration of Pococapsule mimics the wiring done by hand in *Appendix E ApplicationBuilder: Complete Source Code*.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE poco-application-context SYSTEM "http://www.pocomatic.com/poco-application-context.dtd">
<poco-application-context>
<!-- Game Application -->
<bean id="Game"
class="Game"
destroy-method="delete">
<method-arg ref="GraphicFrameWork" />
<method-arg ref="TerrainManager" />
<method-arg ref="ScoreComponent" />
<method-arg ref="MenuComponent" />
<method-arg ref="HighScoreComponent" />
<method-arg ref="CreditComponent" />
<method-arg ref="LevelManager" />
<method-arg ref="GameObjectManager" />
<method-arg ref="GameObjectFactory" />
<method-arg ref="PhysicsComponent" />
<method-arg ref="CameraView" />
<method-arg ref="TextFont" />
</bean>
<!-- END Game Application -->

<!-- HGE Setup -->
<bean id="GraphicFrameWork"
class="HGE"
factory-method="InitGraphicFrameWork"
destroy-method="ShutDownGraphicFrameWork"
        >
<method-arg type="cstring" value="Black Lagoon, the Adventures of Rock"/>
<method-arg ref="updateGame"/>
<method-arg ref="renderGame"/>
<method-arg type="short" value="800"/>
<method-arg type="short" value="600"/>
</bean>
<bean id="updateGame" class="*hgeCallback"
factory-method="GetUpdateGameMethod" />
<bean id="renderGame" class="*hgeCallback"
factory-method="GetRenderGameMethod" />
<!-- END HGE Setup -->
<!-- Background -->
<bean id="BackgroundSprite"
class="hgeSprite"
destroy-method="delete"
        >
<method-arg>
<bean id="BackgroundTexture"
class="*HTEXTURE"
```

```
factory-bean="GraphicFrameWork"
factory-method="Texture_Load"
destroy-bean="GraphicFrameWork"
destroy-method="Texture_Free"
        >
<method-arg type="cstring" value="textures/background_texture.tga" />
</bean>
</method-arg>
<method-arg type="float" value="0.0" />
<method-arg type="float" value="0.0" />
<method-arg type="float" value="800.0" />
<method-arg type="float" value="600.0" />
<ioc method="SetBlendMode">
<method-arg type="short" value="BLEND_BLEND_COLORMUL | BLEND_ALPHABLEND |
BLEND_NOZWRITE)" />
</ioc>
<ioc method="SetColor">
<method-arg ref="FontColor" />
</ioc>
</bean>
<!-- END Background -->

<!-- Score Component -->
<bean id="ScoreComponent"
class="ScoreComponent"
destroy-method="delete"
        >
<method-arg ref="ScoreComponentBackgroundSprite" />
<method-arg ref="ScoreComponentFont" />
</bean>
<bean id="ScoreComponentBackgroundSprite"
class="hgeSprite"
destroy-method="delete"
        >
<method-arg ref="ScoreComponentTexture" />
<method-arg type="float" value="0.0" />
<method-arg type="float" value="0.0" />
<method-arg type="float" value="200.0" />
<method-arg type="float" value="600.0" />
</bean>
<bean id="ScoreComponentTexture"
class="*HTEXTURE"
factory-bean="GraphicFrameWork"
factory-method="Texture_Load"
destroy-bean="GraphicFrameWork"
destroy-method="Texture_Free"
        >
<method-arg type="cstring" value="textures/score_texture.tga" />
</bean>
<bean id="ScoreComponentFont"
class="hgeFont"
destroy-method="delete"
        >
```

```
<method-arg type="string" value="fonts/BankGothic_Lt_BT_24.fnt" />
<ioc method="SetScale">
<method-arg type="float" value="1.0" />
</ioc>
<ioc method="SetColor">
<method-arg ref="ScoreFontColor" />
</ioc>
</bean>
<!-- END Score Component -->

<!-- Menu Component -->
<bean id="MenuComponent"
class="MenuComponent"
destroy-method="delete"
        >
<method-arg ref="MenuComponentBackgroundSprite" />
<method-arg ref="TitleFont" />
<method-arg ref="TitleFontShadow" />
<method-arg ref="TitleFontSelect" />

</bean>
<bean id="MenuComponentBackgroundSprite"
class="hgeSprite"
destroy-method="delete"
        >
<method-arg ref="MenuComponentTexture" />
<method-arg type="float" value="0.0" />
<method-arg type="float" value="0.0" />
<method-arg type="float" value="800.0" />
<method-arg type="float" value="600.0" />
</bean>
<bean id="MenuComponentTexture"
class="*HTEXTURE"
factory-bean="GraphicFrameWork"
factory-method="Texture_Load"
destroy-bean="GraphicFrameWork"
destroy-method="Texture_Free"
                    >
<method-arg type="cstring" value="textures/menu_texture.tga" />
</bean>
<!-- END Menu Component -->

<!-- High Score Component -->
<bean id="HighScoreComponent"
class="HighScoreComponent"
destroy-method="delete"
        >
<method-arg ref="BackgroundSprite" />
<method-arg ref="TextFont" />
<method-arg ref="TextFontSelect" />
<method-arg ref="TitleFontSelect" />
<method-arg ref="TitleFontShadow" />
```

```
<method-arg type="string" value="highscore.txt" />
</bean>
<!-- END High Score Component -->

<!--- Credit Component -->
<bean id="CreditComponent"
class="CreditComponent"
destroy-method="delete"
         >
<method-arg ref="BackgroundSprite" />
<method-arg ref="TextFont" />
</bean>
<!--- END Credit Component -->

<!-- Level Manager -->
<bean id="LevelManager"
class="LevelManager"
destroy-method="delete"
         >
<method-arg ref="BackgroundSprite" />
<method-arg ref="TitleFontSelect" />
<method-arg ref="TitleFontShadow" />
<method-arg type="string" value="levels/level.lev" />
<method-arg type="short" value="1" />
</bean>
<!-- END Level Manager-->

<!-- Physics Component -->
<bean id="PhysicsComponent"
class="PhysicsComponent"
destroy-method="delete">
</bean>
<!-- END Physics Component -->

<!-- Game Object Factory Component -->
<bean id="GameObjectFactory"
class="GameObjectFactory"
destroy-method="delete"
         >
<method-arg ref="GameObjectTexture" />
</bean>
<bean id="GameObjectTexture"
class="*HTEXTURE"
factory-bean="GraphicFrameWork"
factory-method="Texture_Load"
destroy-bean="GraphicFrameWork"
destroy-method="Texture_Free"
         >
<method-arg type="cstring" value="textures/game_object_texture_atlas.tga" />
</bean>
<!-- END Game Object Factory Component -->
```

```
<!-- Game Object Manager -->
<bean id="GameObjectManager"
class="GameObjectManager"
destroy-method="delete"
         >
<method-arg ref="GameObjectFactory" />
</bean>
<!-- END Game Object Manager -->

<!-- Terrain Manager -->
<bean id="TerrainManager"
class="TerrainManager"
destroy-method="delete">
<method-arg ref="TerrainTileTypes" />
</bean>
<bean id="TerrainTileTypes"
class="TerrainTileTypes{21}"
destroy-method="delete"
         >
<method-arg ref="TileSprites" />
</bean>
<bean id="TileSprites"
class="TileSprites"
destroy-method="delete"
         >
<method-arg ref="TerrainTexture" />
<method-arg type="float" value="60.0" />
<method-arg type="float" value="60.0" />
<method-arg type="float" value="2.0" />
<method-arg type="float" value="3.0" />
</bean>
<bean id="TerrainTexture"
class="*HTEXTURE"
factory-bean="GraphicFrameWork"
factory-method="Texture_Load"
destroy-bean="GraphicFrameWork"
destroy-method="Texture_Free"
         >
<method-arg type="cstring" value="textures/terrain_texture_atlas.tga" />
</bean>
<!-- END Terrain Manager -->

<!-- Camera View -->
<bean id="CameraView"
class="Rect"
destroy-method="delete"
         >
<method-arg type="float" value="0.0"/>
<method-arg type="float" value="0.0"/>
<method-arg type="float" value="600.0"/>
<method-arg type="float" value="600.0"/>
</bean>
```

```
<!-- END Camera View -->

<!-- Fonts -->
<bean id="TextFont"
class="hgeFont"
destroy-method="delete"
        >
<method-arg type="cstring" value="fonts/BankGothic_Md_BT_20.fnt" />
<ioc method="SetScale">
<method-arg type="float" value="1.0" />
</ioc>
<ioc method="SetColor">
<method-arg ref="FontColor" />
</ioc>
</bean>
<bean id="TextFontSelect"
class="hgeFont"
destroy-method="delete"
        >
<method-arg type="cstring" value="fonts/BankGothic_Md_BT_20.fnt" />
<ioc method="SetColor">
<method-arg ref="FontSelectColor" />
</ioc>
<ioc method="SetScale">
<method-arg type="float" value="1.0" />
</ioc>
</bean>
<bean id="TitleFont"
class="hgeFont"
destroy-method="delete"
        >
<method-arg type="cstring" value="fonts/BankGothic_Lt_BT_64.fnt" />
<ioc method="SetScale">
<method-arg type="float" value="0.9" />
</ioc>
<ioc method="SetColor">
<method-arg ref="FontColor" />
</ioc>
</bean>
<bean id="TitleFontSelect"
class="hgeFont"
destroy-method="delete"
        >
<method-arg type="cstring" value="fonts/BankGothic_Lt_BT_64.fnt" />
<ioc method="SetScale">
<method-arg type="float" value="0.9" />
</ioc>
<ioc method="SetColor">
<method-arg ref="FontSelectColor" />
</ioc>
</bean>
<bean id="TitleFontShadow"
```

```
class="hgeFont"
destroy-method="delete"
        >
<method-arg type="cstring" value="fonts/BankGothic_Lt_BT_64.fnt" />
<ioc method="SetScale">
<method-arg type="float" value="1.0" />
</ioc>
<ioc method="SetColor">
<method-arg ref="FontShadowColor" />
</ioc>
</bean>
<!-- END Fonts -->

<!-- Colors -->
<bean id="FontColor"
class="*DWORD"
factory-method="ARGB">
<method-arg type="ulong" value="250" />
<method-arg type="ulong" value="170" />
<method-arg type="ulong" value="170" />
<method-arg type="ulong" value="160" />
</bean>

<bean id="ScoreFontColor"
class="*DWORD"
factory-method="ARGB">
<method-arg type="ulong" value="255" />
<method-arg type="ulong" value="250" />
<method-arg type="ulong" value="250" />
<method-arg type="ulong" value="166" />
</bean>
<bean id="FontSelectColor"
class="*DWORD"
factory-method="ARGB">
<method-arg type="ulong" value="238" />
<method-arg type="ulong" value="145" />
<method-arg type="ulong" value="21" />
<method-arg type="ulong" value="23" />
</bean>
<bean id="FontShadowColor"
class="*DWORD"
factory-method="ARGB">
<method-arg type="ulong" value="160" />
<method-arg type="ulong" value="10" />
<method-arg type="ulong" value="10" />
<method-arg type="ulong" value="10" />
</bean>
<!-- End Colors -->

</poco-application-context>
```

Appendix D Autumn Configuration XML

This configuration follows the Autumn Framework syntax and mimics the behaviour of the hand wiring done in *Appendix E ApplicationBuilder: Complete Source Code*.

```
<?xml version="1.0" encoding="utf-8"?>
<autumn xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="autumn.xsd">
  <library path="local">
    <beans>
      <bean class="Game" name="GameApp">
        <constructor-arg>
          <argument type="HGE">
            <ref bean="gfw"/>
          </argument>
          <argument type="TerrainManager">
            <ref bean="TerrainMnager"/>
          </argument>
          <argument type="ScoreComponent">
            <ref bean="ScoreComp"/>
          </argument>
          <argument type="MenuComponent">
            <ref bean="ScoreComp"/>
          </argument>
          <argument type="HighScoreComponent">
            <ref bean="HighScoreComp"/>
          </argument>
          <argument type="CreditComponent">
            <ref bean="CreditComp"/>
          </argument>
          <argument type="LevelManager">
            <ref bean="LevelManager"/>
          </argument>
          <argument type="GameObjectManager">
            <ref bean="GameObjectManager"/>
          </argument>
          <argument type="GameObjectFactory">
            <ref bean="GameObjectFactory"/>
          </argument>
          <argument type="PhysicsComponent">
            <ref bean="PhysicsComp"/>
          </argument>
          <argument type="Rect">
            <ref bean="CameraView"/>
          </argument>
          <argument type="hgeFont">
            <ref bean="TextFont"/>
          </argument>
        </constructor-arg>
      </bean>

      <bean class="GraphicsFrameWork"
            name="GFWBuilder"
            singleton="true"
            >
      </bean>

      <bean class ="HGE"
        name="gfw"
        factory-bean="GFWBuilder"
        factory-method="InitGraphicFrameWork"
        destroy-method="ShutDownGraphicFrameWork"
        >
        <constructor-arg>
          <argument>
            <value>Black Lagoon, the Adventures of Rock</value>
          </argument>
          <argument>
```

```
      <value>800</value>
    </argument>
    <argument>
      <value>600</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="HTEXTURE"
      name="BackgroundTexture"
      factory-bean="gfw"
      factory-method="Texture_Load"
      destroy-method="Texture_Free"
      singleton="true">
  <constructor-arg>
    <argument>
      <value>textures/background_texture.tga</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="hgeSprite" name="BackgroundSprite" singleton="true">
  <constructor-arg>
    <argument type="HTEXTURE">
      <ref bean="BackgroundTexture"/>
    </argument>
  </constructor-arg>
</bean>

<bean class="PhysicsComponent" name="PhysicsComp" />

<bean class="TerrainManager" name="TerrainManager">
  <constructor-arg>
    <argument type="TerrainTileTypes{21}">
      <ref bean="Types"/>
    </argument>
  </constructor-arg>
</bean>

<bean class="TerrainTileTypes{21}" name="TileTypes">
  <constructor-arg>
    <argument>
      <ref bean="TileSprites"/>
    </argument>
  </constructor-arg>
</bean>

<bean class="TileSprites" name="TileSprites">
  <constructor-arg>
    <argument>
      <ref bean="TerrainTexture"/>
    </argument>
  </constructor-arg>
</bean>

<bean class="HTEXTERE"
      name="TerrainTexture"
      factory-bean="gfw"
      factory-method="Texture_Load"
      destroy-method="Texture_Free"
      singleton="true">
  <constructor-arg>
    <argument type="char*">
      <value>textures/terrain_texture_atlas.tga</value>
    </argument>
  </constructor-arg>
</bean>
```

```
<bean class="ScoreComponet" name="ScoreComp">
  <constructor-arg>
    <argument type="hgeSprite">
      <ref bean="ScoreComponentBackgroundSprite"/>
    </argument>
    <argument type="hgeFont">
      <ref bean="ScoreComponentFont"/>
    </argument>
  </constructor-arg>
</bean>

<bean class="hgeSprite" name="ScoreComponentSprite">
  <constructor-arg>
    <argument type="HTEXTURE">
      <ref bean="ScoreComponentTexture"/>
    </argument>
    <argument type="float">
      <value>0.0</value>
    </argument>
    <argument type="float">
      <value>0.0</value>
    </argument>
    <argument type="float">
      <value>200.0</value>
    </argument>
    <argument type="float">
      <value>600.0</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="HTEXTERE"
      name="ScoreComponentTexture"
      factory-bean="gfw"
      factory-method="Texture_Load"
      destroy-method="Texture_Free"
      singleton="true">
  <constructor-arg>
    <argument type="char*">
      <value>textures/score_texture.tga</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="hgeFont" name="ScoreComponentFont">
  <constructor-arg>
    <argument>
      <value>fonts/BankGothic_Lt_BT_24.fnt</value>
    </argument>
  </constructor-arg>
  <properties>
    <property name="SetScale" type="float">
      <value>1.0</value>
    </property>
    <property name="SetColor" type="DWORD">
      <ref bean="ScoreFontColor"/>
    </property>
  </properties>
</bean>

<bean class="DWORD" name="ScoreFontColor"
factory-method="ARGB">
  <constructor-arg>
    <argument type="ulong">
      <value>255</value>
    </argument>
    <argument type="ulong">
      <value>250</value>
```

```
    </argument>
    <argument type="ulong">
      <value>250</value>
    </argument>
    <argument type="ulong">
      <value>166</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="MenuComponent" name="MenuComp">
  <constructor-arg>
    <argument>
      <ref bean="MenuComponentTexture"/>
    </argument>
    <argument type="float">
      <value>0.0</value>
    </argument>
    <argument type="float">
      <value>0.0</value>
    </argument>
    <argument type="float">
      <value>800.0</value>
    </argument>
    <argument type="float">
      <value>600.0</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="HTEXTERE"
      name="MenuComponentTexture"
      factory-bean="gfw"
      factory-method="Texture_Load"
      destroy-method="Texture_Free"
      singleton="true">
  <constructor-arg>
    <argument type="char*">
      <value>textures/menu_texture.tga</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="HighScoreComponent" name="HighScoreComp">
  <constructor-arg>
    <argument type="hgeSprite">
      <ref bean="BackgroundSprite"/>
    </argument>
    <argument type="hgeFont">
      <ref bean="TextFont"/>
    </argument>
    <argument type="hgeFont">
      <ref bean="TextFontSelect"/>
    </argument>
    <argument type="hgeFont">
      <ref bean="TitleFontSelect"/>
    </argument>
    <argument>
      <ref bean="TitleFontShadow"/>
    </argument>
    <argument type="char*">
      <value>highscore.txt</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="CreditComponet" name="CreditComp">
  <constructor-arg>
```

```
    <argument type="hgeSprite">
      <ref bean="BackgroundSprite"/>
    </argument>
    <argument type="hgeFont">
      <ref bean="TextFont"/>
    </argument>
  </constructor-arg>
</bean>

<bean class="LevelManager" name="LevelManager">
  <constructor-arg>
    <argument type="hgeSprite">
      <ref bean="BackgroundSprite"/>
    </argument>
    <argument type="hgeFont">
      <ref bean="TitleFontSelect"/>
    </argument>
    <argument type="hgeFont">
      <ref bean="TitleFontShadow"/>
    </argument>
    <argument type="string">
      <value>levels/level.lev</value>
    </argument>
    <argument type="short">
      <value>1</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="GameObjectFactory" name="GameObjectFactory">
  <constructor-arg>
    <argument type="hgeSprite">
      <ref bean="GameObjectTexture"/>
    </argument>
  </constructor-arg>
</bean>

<bean class="HTEXTERE"
      name="GameObjectTexture"
      factory-bean="gfw"
      factory-method="Texture_Load"
      destroy-method="Texture_Free"
      singleton="true">
  <constructor-arg>
    <argument type="char*">
      <value>textures/game_object_texture_atlas.tga</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="GameObjectManager" name="GameObjectManager">
  <constructor-arg>
    <argument type="GameObjectFactory">
      <ref bean="GameObjectFactory"/>
    </argument>
  </constructor-arg>
</bean>

<bean class="Rect" name="CameraView">
  <constructor-arg>
    <argument type="float">
      <value>0.0</value>
    </argument>
    <argument type="float">
      <value>0.0</value>
    </argument>
    <argument type="float">
      <value>600.0</value>
```

```
    </argument>
    <argument type="float">
      <value>600.0</value>
    </argument>
  </constructor-arg>
</bean>

<bean class="hgeFont" name="TextFont">
  <constructor-arg>
    <argument>
      <value>fonts/BankGothic_Md_BT_20.fnt</value>
    </argument>
  </constructor-arg>
  <properties>
    <property name="SetScale" type="float">
      <value>1.0</value>
    </property>
    <property name="SetColor" type="DWORD">
      <ref bean="FontColor"/>
    </property>
  </properties>
</bean>

<bean class="hgeFont" name="TextFontSelect">
  <constructor-arg>
    <argument>
      <value>fonts/BankGothic_Md_BT_20.fnt</value>
    </argument>
  </constructor-arg>
  <properties>
    <property name="SetScale" type="float">
      <value>1.0</value>
    </property>
    <property name="SetColor" type="DWORD">
      <ref bean="FontColor"/>
    </property>
  </properties>
</bean>

<bean class="hgeFont" name="TitleFont">
  <constructor-arg>
    <argument>
      <value>fonts/BankGothic_Lt_BT_64.fnt</value>
    </argument>
  </constructor-arg>
  <properties>
    <property name="SetScale" type="float">
      <value>0.9</value>
    </property>
    <property name="SetColor" type="DWORD">
      <ref bean="FontColor"/>
    </property>
  </properties>
</bean>

<bean class="hgeFont" name="TitleFontSelect">
  <constructor-arg>
    <argument>
      <value>fonts/BankGothic_Lt_BT_64.fnt</value>
    </argument>
  </constructor-arg>
  <properties>
    <property name="SetScale" type="float">
      <value>0.9</value>
    </property>
    <property name="SetColor" type="DWORD">
      <ref bean="FontSelectColor"/>
    </property>
```

```
      </properties>
    </bean>

    <bean class="hgeFont" name="TitleFontShadow">
      <constructor-arg>
        <argument>
          <value>fonts/BankGothic_Lt_BT_64.fnt</value>
        </argument>
      </constructor-arg>
      <properties>
        <property name="SetScale" type="float">
          <value>1.0</value>
        </property>
        <property name="SetColor" type="DWORD">
          <ref bean="FontShadowColor"/>
        </property>
      </properties>
    </bean>

    <bean class="DWORD" name="FontColor"
          factory-method="ARGB">
      <constructor-arg>
        <argument type="ulong">
          <value>160</value>
        </argument>
        <argument type="ulong">
          <value>10</value>
        </argument>
        <argument type="ulong">
          <value>10</value>
        </argument>
        <argument type="ulong">
          <value>10</value>
        </argument>
      </constructor-arg>
    </bean>

    <bean class="FileLogger" name="AutumnFrameworkLog" singleton="true">
      <constructor-arg>
        <argument>
          <value>AutumnApp.log</value>
        </argument>
        <argument>
          <value>4</value>
        </argument>
      </constructor-arg>
    </bean>
  </beans>
 </library>
</autumn>
```

Appendix E ApplicationBuilder: Complete Source Code

Here follows the complete source code for the ApplicationBuilder class. The class was used to manually wire Codebase II's components together and serves as an example of how dependency injection can be used to separate instantiation and building of object graph from the domain logic.

The IoC container configuration of PocoCapsule (see Appendix CPococapsule Configuration XML) mimics the behaviour of the Application Builder.

ApplicationBuilder.h

```
#pragma once
#include "ITerrainManager.h"
#include "ITerrainTileTypes.h"
#include "TileSprites.h"
#include "ScoreComponent.h"
#include "MenuComponent.h"
#include "HighScoreComponent.h"
#include "CreditComponent.h"
#include "LevelManager.h"
#include "GameObjectManager.h"
#include "GameObjectFactory.h"
#include "PhysicsComponent.h"
#include "game.h"
```

```
class ApplicationBuilder
{
public:
   ApplicationBuilder(HGE* hge);
   ~ApplicationBuilder(void);

   Game* buildGame();
```

```
private:
   void LoadTextures(HGE* gfw);

   ScoreComponent* buildScoreComponent(HTEXTURE texture, const char*
fontFile);
   MenuComponent* buildMenuComponent(HTEXTURE texture, const char*
fontFile);
   HighScoreComponent* buildHighScoreComponent(HTEXTURE texture,
      const char* fontFile,
      const char* fontTitleFile,
      const char* highScoreFile);

   CreditComponent* buildCreditComponent(HTEXTURE texture, const char*
fontFile);
   ITerrainManager* buildTerrainManager(HTEXTURE tileTextureAtlas);
   LevelManager* buildLevelManager(HTEXTURE texture, const char* filename,
int startingLevel);
```

```
private:
   HGE* gfw;

   HTEXTURE terrainTexture;
   HTEXTURE scoreComponentTexture;
   HTEXTURE menuTexture;
   HTEXTURE backgroundTexture;
   HTEXTURE gameObjectTexture;

   ScoreComponent* scoreComponent;
   MenuComponent* menuComponent;
   HighScoreComponent* highScoreComponent;
   CreditComponent* creditComponent;
   LevelManager* levelManager;
   GameObjectFactory* gameObjectFactory;
   GameObjectManager* gameObjectManager;
   PhysicsComponent* physicsComponent;

   hgeFont* font;

   hgeSprite* scoreComponentBackground;
   hgeFont* scoreComponentFont;

   hgeSprite* menuComponentBackground;
   hgeFont* menuComponentFont;
   hgeFont* menuComponentFontShadow;
   hgeFont* menuComponentFontSelected;

   hgeSprite* highScoreComponentBackground;
   hgeFont* highScoreComponentFont;
   hgeFont* highScoreComponentFontSelected;
   hgeFont* highScoreComponentFontTitle;
   hgeFont* highScoreComponentFontShadow;

   hgeSprite* creditComponentBackground;
   hgeFont* creditComponentFont;

   ITerrainManager* terrainManager;
   ITerrainTileTypes* terrainTileTypes;
   ITileSprites* tileSprites;

   hgeSprite* levelManagerBackground;
   hgeFont* levelManagerFont;
   hgeFont* levelManagerFontShadow;
};
```

ApplicationBuilder.cpp

```
#include "ApplicationBuilder.h"
#include "TerrainTileTypes.h"
#include "TileSprites.h"
#include "TerrainManager.h"
#define NR_OF_TERRAIN_TILE_TYPES 21
```

```
ApplicationBuilder::ApplicationBuilder(HGE* hge) :
   gfw(hge),
   scoreComponentBackground(0),
   scoreComponentFont(0),
   menuComponentBackground(0),
   menuComponentFont(0),
   menuComponentFontShadow(0),
   menuComponentFontSelected(0),
   highScoreComponentBackground(0),
   highScoreComponentFont(0),
   highScoreComponentFontSelected(0),
   highScoreComponentFontTitle(0),
   highScoreComponentFontShadow(0),
   creditComponentBackground(0),
   creditComponentFont(0),
   levelManagerBackground(0),
   levelManagerFont(0),
   levelManagerFontShadow(0),
   terrainManager(0),
   terrainTileTypes(0),
   tileSprites(0)
{
}
```

```
ApplicationBuilder::~ApplicationBuilder(void)
{
   SAFE_DELETE(scoreComponent);
   SAFE_DELETE(menuComponent);
   SAFE_DELETE(highScoreComponent);
   SAFE_DELETE(creditComponent);
   SAFE_DELETE(levelManager);
   SAFE_DELETE(gameObjectManager);
   SAFE_DELETE(gameObjectFactory);
   SAFE_DELETE(physicsComponent);
   SAFE_DELETE(scoreComponentBackground);
   SAFE_DELETE(scoreComponentFont);
   SAFE_DELETE(menuComponentBackground);
   SAFE_DELETE(menuComponentFont);
   SAFE_DELETE(menuComponentFontShadow);
   SAFE_DELETE(menuComponentFontSelected);
   SAFE_DELETE(highScoreComponentBackground);
   SAFE_DELETE(highScoreComponentFont);
   SAFE_DELETE(highScoreComponentFontSelected);
   SAFE_DELETE(highScoreComponentFontTitle);
   SAFE_DELETE(highScoreComponentFontShadow);
   SAFE_DELETE(creditComponentBackground);
   SAFE_DELETE(creditComponentFont);
   SAFE_DELETE(terrainManager);
   SAFE_DELETE(terrainTileTypes);
   SAFE_DELETE(tileSprites);
   SAFE_DELETE(levelManagerBackground);
   SAFE_DELETE(levelManagerFont);
   SAFE_DELETE(levelManagerFontShadow);
   // Release textures
   if (terrainTexture != 0)
      gfw->Texture_Free(terrainTexture);
   if (scoreComponentTexture != 0)
      gfw->Texture_Free(scoreComponentTexture);
   if (menuTexture != 0)
      gfw->Texture_Free(menuTexture);
   if (backgroundTexture != 0)
      gfw->Texture_Free(backgroundTexture);
   if (gameObjectTexture != 0)
      gfw->Texture_Free(gameObjectTexture);
}
```

```
Game* ApplicationBuilder::buildGame()
{
   LoadTextures(this->gfw);

   Rect cameraView(Rect(0.0f,0.0f,600.0f,600.0f));

   tileSprites = new TileSprites(terrainTexture,
      60.0f, 60.0f, 2.0f, 3.0f);
   terrainTileTypes = new TerrainTileTypes(tileSprites);
   terrainManager = new TerrainManager(terrainTileTypes);

   scoreComponent = buildScoreComponent(scoreComponentTexture,
      "fonts/BankGothic_Lt_BT_24.fnt");
   menuComponent = buildMenuComponent(menuTexture,
      "fonts/BankGothic_Lt_BT_64.fnt");
   highScoreComponent = buildHighScoreComponent(
      backgroundTexture,
      "fonts/BankGothic_Md_BT_20.fnt",
      "fonts/BankGothic_Lt_BT_64.fnt",
      "highscore.txt");
   creditComponent = buildCreditComponent(backgroundTexture,
      "fonts/BankGothic_Lt_BT_20.fnt");
   levelManager = buildLevelManager(backgroundTexture,
      "levels/level.lev", 1);
   gameObjectFactory  = new GameObjectFactory(gameObjectTexture);
   gameObjectManager = new GameObjectManager(gameObjectFactory);

   physicsComponent = new PhysicsComponent();

   font = new hgeFont("fonts/BankGothic_Lt_BT_20.fnt");
   font->SetScale(1.0f);
   font->SetColor(ARGB(0xFF,0xFA,0xFA,0xA6));

   Game* game = new Game(gfw,
      terrainManager,
      scoreComponent,
      menuComponent,
      highScoreComponent,
      creditComponent,
      levelManager,
      gameObjectManager,
      gameObjectFactory,
      physicsComponent,
      cameraView,
      font);
   return game;
}
```

```
void ApplicationBuilder::LoadTextures(HGE* gfw)
{
   terrainTexture = gfw->Texture_Load(
      "textures/terrain_texture_atlas.tga");;
   scoreComponentTexture = gfw->Texture_Load(
      "textures/score_texture.tga");;
   menuTexture = gfw->Texture_Load("textures/menu_texture.tga");;
   backgroundTexture = gfw->Texture_Load(
      "textures/background_texture.tga");
   gameObjectTexture = gfw->Texture_Load(
      "textures/game_object_texture_atlas.tga");
}
```

```
ScoreComponent* ApplicationBuilder::buildScoreComponent(HTEXTURE texture,
const char* fontFile)
{
   scoreComponentBackground = new hgeSprite(texture, 0.0f, 0.0f, 200,
SCREEN_HEIGHT);
   scoreComponentFont = new hgeFont(fontFile);
   scoreComponentFont->SetScale(1.0f);
   scoreComponentFont->SetColor(ARGB(0xFF,0xFA,0xFA,0xA6));
   return new ScoreComponent(scoreComponentBackground,
scoreComponentFont);
}
```

```
MenuComponent* ApplicationBuilder::buildMenuComponent(HTEXTURE texture,
const char* fontFile)
{
   menuComponentBackground = new hgeSprite(texture, 0.0f, 0.0f,
SCREEN_WIDTH, SCREEN_HEIGHT);

   menuComponentFont = new hgeFont(fontFile);
   menuComponentFont->SetScale(0.8f);
   menuComponentFont->SetColor(ARGB(0xA0,0xAF,0xAF,0xA0));

   menuComponentFontShadow = new hgeFont(fontFile);
   menuComponentFontShadow->SetScale(1.0f);
   menuComponentFontShadow->SetColor(ARGB(0xA0,0x10,10,10));

   menuComponentFontSelected = new hgeFont(fontFile);
   menuComponentFontSelected->SetScale(0.8f);
   menuComponentFontSelected->SetColor(ARGB(0xEE,145,21,23));

   return new MenuComponent(menuComponentBackground, menuComponentFont,
menuComponentFontShadow, menuComponentFontSelected);
}
```

```
HighScoreComponent*
ApplicationBuilder::buildHighScoreComponent(HTEXTURE texture,
      const char* fontFile,
      const char* fontTitleFile,
      const char* highScoreFile
   )
{
   highScoreComponentBackground = new hgeSprite(texture, 0.0f, 0.0f,
SCREEN_WIDTH, SCREEN_HEIGHT);
   highScoreComponentBackground->SetBlendMode(BLEND_COLORMUL |
BLEND_ALPHABLEND | BLEND_NOZWRITE);
   highScoreComponentBackground->SetColor(ARGB(255, 100,100,100));

   highScoreComponentFont = new hgeFont(fontFile);
   highScoreComponentFont->SetScale(1.0f);
   highScoreComponentFont->SetColor(ARGB(0xFA,0xAA,0xAA,0xA0));

   highScoreComponentFontSelected = new hgeFont(fontFile);
   highScoreComponentFontSelected->SetScale(1.0f);
   highScoreComponentFontSelected->SetColor(ARGB(0xEE,145,21,23));

   highScoreComponentFontTitle = new hgeFont(fontTitleFile);
   highScoreComponentFontTitle->SetScale(0.9f);
   highScoreComponentFontTitle->SetColor(ARGB(0xEE,145,21,23));

   highScoreComponentFontShadow = new hgeFont(fontTitleFile);
   highScoreComponentFontShadow->SetScale(1.0f);
   highScoreComponentFontShadow->SetColor(ARGB(0xA0,0x10,10,10));

   return new HighScoreComponent(highScoreComponentBackground,
      highScoreComponentFont,
      highScoreComponentFontSelected,
      highScoreComponentFontTitle,
      highScoreComponentFontShadow,
      highScoreFile
      );
}
```

```
CreditComponent*
ApplicationBuilder::buildCreditComponent(
      HTEXTURE texture,
      const char* fontFile
   )
{
   creditComponentBackground = new hgeSprite(texture, 0.0f, 0.0f,
SCREEN_WIDTH, SCREEN_HEIGHT);
   creditComponentBackground->SetBlendMode(BLEND_COLORMUL |
BLEND_ALPHABLEND | BLEND_NOZWRITE);
   creditComponentBackground->SetColor(ARGB(255, 100,100,100));

   creditComponentFont = new hgeFont(fontFile);
   creditComponentFont->SetScale(1.0f);
   creditComponentFont->SetColor(ARGB(0xFA,0xAA,0xAA,0xA0));

   return new CreditComponent(creditComponentBackground,
creditComponentFont);
}
```

```
LevelManager* ApplicationBuilder::buildLevelManager(HTEXTURE texture,
const char* filename, int startingLevel)
{
   levelManagerBackground = new hgeSprite(texture, 0.0f, 0.0f,
      SCREEN_WIDTH, SCREEN_HEIGHT);
   levelManagerBackground->SetBlendMode(BLEND_COLORMUL |
      BLEND_ALPHABLEND | BLEND_NOZWRITE);
   levelManagerBackground->SetColor(ARGB(255, 100,100,100));

   levelManagerFont = new hgeFont("fonts/BankGothic_Lt_BT_64.fnt");
   levelManagerFont->SetScale(0.9f);
   levelManagerFont->SetColor(ARGB(0xEE,145,21,23));

   levelManagerFontShadow = new hgeFont("fonts/BankGothic_Lt_BT_64.fnt");
   levelManagerFontShadow->SetScale(1.0f);
   levelManagerFontShadow->SetColor(ARGB(0xA0,0x10,10,10));

   return new LevelManager(levelManagerBackground, levelManagerFont,
levelManagerFontShadow,
      filename, startingLevel);
}
```