DEGREE PROJECT 2003:PM01

Division of Computer Science at the Department of Informatics and Mathematics

Design Patterns in Practice

Ingemar Jacobsson

HÖGSKOLAN
TROLLHÄTTAN • UDDEVALLA

DEGREE PROJECT

University of Trollhättan · Uddevalla
Department of Informatics and Mathematics

Degree project for master degree in Software engineering

Design Patterns in Practice

Ingemar Jacobsson

Examiner:	
Dr. Steven Kirk	Department of Informatics and Mathematics
Supervisor:	
Andreas Boklund	Department of Informatics and Mathematics

Trollhättan, 2003

2003:PM01

EXAMENSARBETE

Design Patterns in Practice

Ingemar Jacobsson

Sammanfattning

Grundläggande för alla vetenskapsdiscipliner är en gemensam begreppsvärld för att beskriva vetenskapsdisciplinens koncept och ett språk för att sammanbinda dessa koncept. Software Engineering är en relativt ung vetenskapsdisciplin, som fortfarande söker ett gemensamt språk som möjliggör en effektiv kommunikation mellan de enskilda utövarna. Målet för Design Patterns är att skapa en gemensam begreppsvärld som underlättar arbetet med att lösa återkommande designproblem vid utövningen av objektorienterad programmering. Design Patterns utgör det språk som behövs för att kommunicera insikter och erfarenheter om hur återkommande designproblem kan lösas. Med hjälp av detta språk kan utövarna diskutera vilken lösning som är tillämplig och eftersträvansvärd, men också vilka effekter som ett tillämpat Design Pattern medför. Tillämpningen av Design Patterns medför också att goda designlösningar kan bevaras och återanvändas.

Syftet med detta examensarbete är att utvärdera konceptet Design Patterns, genom att tillämpa fem av de Design Patterns, som beskrivs av Gamma et. al i boken ”Design Patterns”, på en applikation och sedan värdera hur väl den resulterande applikationen går att återanvända och underhålla.

Utgivare:	Högskolan Trollhättan · Uddevalla, Institutionen för Informatik och Matematik Box 957, 461 29 Trollhättan Tel: 0520-47 50 00 Fax: 0520-47 50 99		
Examinator:	Dr. Steven Kirk		
Handledare:	Andreas Boklund, HTU		
Huvudämne:	Programvaruteknik	**Språk**:	Engelska
Nivå:	Fördjupningsnivå 2	**Poäng**:	10
Rapportnr:	2003:PM01	**Datum**:	2003-05-20
Nyckelord:	Object-oriented programming, Design Patterns, reusability, maintainability		

Förord

Jag vill tacka Dr. Steven Kirk för hans outtröttliga stöd i samband med detta examensarbete. Vi har haft många diskussioner, som i stor utsträckning bidragit till den slutliga utformningen av detta examensarbete. Jag vill även rikta min tacksamhet till Dr. Ludwik Kuzniarz, som väckte mitt intresse för Design Patterns.

Innehållsförteckning

Nomenklatur

Kvantitativa mått:

CC = Cyclomatic Complexity

A = Abstractness

Ca = Afferent Couplings

Ce = Efferent Coupling

I = Instability

Benämningen av C#-projekt (CD-media):

F = FactoryExample

F/B = FactoryBuilderExample

F/B/C = FactoryBuilderCompositeExample

F/B/C/S/O= FactoryBuilderCompositeSingletonObserverExample

Benämningen av C#-komponenter:

CF = Client Form

F pack = Factory package

SC pack = StringChecker package

B pack = Builder package

C pack = Composite package

Övrigt:

JSP = Jackson Structured Programming

UML = Unified Modelling Language

DP = Design Patterns

SSN = Social Security Number, personnummer

EXAMENSARBETE

Design Patterns in Practice

List of Authors
Ingemar Jacobsson
Address
Communicating author: ingemar.jacobsson@telia.com

Abstract

Design Patterns capture solutions in object-oriented programming that have developed and evolved over time. One of the main reasons that computer science researchers began to recognize design patterns is to satisfy this need for elegant, but simple, reusable solutions. In fact, design patterns are a body of literature that forms a language for conveying the structures and mechanisms of object-oriented architectures and allows us to intelligibly reason about the resulting composition and applications.

Design Patterns are a common way to organize objects and the collaborations between objects in object-oriented programs to make those programs easier to reuse and maintain. This formation of a common language between practitioners in object-oriented program is vital for software engineering as a scientific discipline.

The vocabulary and strategies of object-oriented programming such as abstraction, separation, composition and generalization needs a common language to relate these concepts together.

Software engineering, which comprises object-oriented programming, is a relatively young scientific discipline, still struggling to find a shared language that enables an effortless communication between practitioners. Design Patterns satisfies these objectives and help software developers resolve and discuss recurring object-oriented problems encountered throughout all the software development process.

This thesis demonstrates that Design Patterns has the potential to form a shared language for communicating experience and findings about these problems and their solutions.

Codifying these solutions and their collaborations enables object-oriented programmers to achieve sound engineering architecture and design. Design Patterns therefore constitutes a solid foundation and a language to describe good designs and capture experience, in resolving recurring problems in object-oriented programming. This accumulated experience, when codified, facilitates practitioners in object-oriented programming to reuse this solid experience.

The aim of this thesis work is to evaluate the concept of Design Patterns, in terms of the resulting degree of reusability and maintainability of an experimental application. Five Design Patterns, that Gamma et. al describes in the book "Design Patterns", are evaluated. The experimental application is a text field parser component in the programming language C#.

1. Introduction

Software engineering, compared to traditional engineering sciences, is an immature scientific discipline. As an applied science, many IT-projects face problems like budget overruns and delay of delivery [1]. One possible factor that causes software projects to fail could be the lack of firmly established standards. This thesis concerns one candidate standard - design patterns - that could enable a higher degree of reusability and maintainability software and software components. A wider use and reuse of design patterns and components could have a positive effect on cost effectiveness as well as the timeframe predictability for any given IT-project.

The general principle behind the formation and publication of design patterns is to make earlier programming efforts and success obtainable for a wider circle of practitioners in the field of object oriented programming. One way to describe design patterns is that each pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it. Beginners as well as experienced programmers can benefit from colleagues previous work and therefore attain a higher degree of cost effectiveness.

Earlier and somewhat similar endeavours can be found in procedural programming as a result of Böhm's and Jacopini's[2] work in the 1960's. These men proved and defined the fundamental control constructs of structured programs. With some refinement from Edsger Dijkstra [3] and Michael Jackson [4] a firmly established standard on how to design effective, maintainable and readable procedural programs was born, e.g. Jackson Structured Programming (JSP). This work is still valid, above all in designing algorithms, but it is obviously not sufficient – nor was it intended to be so - for object-oriented programming. Techniques and features likes inheritance, interfaces, abstract classes, objects, events and the collaboration between them is somewhat harder to describe than the flow of data through a predictable algorithm. On the other hand, a model of the problem domain is often hard to fully describe with stable algorithms, especially when programming modern graphically intensive software.

The Unified Modelling Language (Object Management Group™, UML) is one of the arisen de facto standards in object oriented analysis and design, that addresses the problem by aiding description of the problem domain in a way that the subsequent system realisation is consistent with the clients need [5]. UML provides a syntax, or more importantly, a mutual language between client and supplier, but it does not have built-in mechanisms to ensure reusability or maintainability as features in the resulting application. To obtain these features, we still have to rely on experienced programming personnel.

Along with wider adoption and adherence to UML and sophisticated modelling tools such as Rational Rose™, BluePrint™, WithClass™ etc, interest for design patterns has increased. The participants in a design pattern are the individual classes, abstract classes, and interfaces that make up the design pattern. Each one assumes a particular role, responsible for carrying out one specific duty. Together, the participants work in concert to accomplish the goal of the design pattern.

1.1 Research question

The primary aim of this work is to evaluate the effects, in terms of reusability and maintainability, of five different design patterns applied on a experimental toolkit component. The effect is studied on how well the resulting design conforms to object-oriented strategies, relevant to the issue, and by using quantitative metrics such as cyclomatic complexity, abstractness and instability.

1.2 Thesis outline

Sections 2 through 4 form the background for the thesis and begin with an overview of object-oriented strategies for reusability and maintainability. Concepts and strategies like abstraction, separation, composition and last but not least generalization, are presented. The latter concept paves the way for design patterns as a possible tool for reusability and maintainability. In Section 3 some quantitative measurements like cyclomatic complexity, abstractness, and instability , which also provide measures of software maintainability and reusability, are described. The background of this thesis ends with Section 4, where the concept of design patterns is defined and the individual design patterns are organized. In Section 5 the test object, the toolkit component, is described in terms of intended use and component requirements. The actual experiment – the design of a toolkit component, using five different design patterns - is presented in Section 6. Finally, Section 6 is closed with a short example on how to expand the experimental application functionality.

2. The objective of Design Patterns and Object-Oriented programming.

Object oriented programming [6] and design patterns [7] have joint objectives. Code, components and designs should be reusable, maintainable and flexible for change. But as [7] states "Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder." (p. 1)

2.1 Object-oriented strategies for reusability and maintainability

In object-oriented programming, four basic strategies in order to achieve maintainability and reusability can be retrieved. These strategies are based on proven and accumulated practical software engineering fundamental experience. The basic strategies in object-oriented programming are abstraction, separation, composition and generalization.

2.2 Strategy: Abstraction

Abstraction is a design technique that focuses on the essential aspects of an entity [8-11]. The less important or non-essential aspects of the entity are ignored or concealed. This feature makes an abstraction an important tool for simplification of a complex situation. The abstraction of an entity can facilitate analysis, experimentation or understanding.

For example, in attempting to classify the elements in the periodic system, early chemists such as Dimitrij Mendelejev applied abstraction to an "element", describing the elements in a way that new elements could be ordered and the properties of the new elements could be predicted with a high degree of certainty. Thus, abstraction ignores a wealth of physical details about each element – the atoms actual diameter, its state of aggregation, etc. However, these other details are not relevant to understanding and modelling the basic periodical system of elements.

Abstraction is concerned with both the attributes and behaviour [8-11]. Attributes refer to the properties or characteristics associated with an entity [8-11]. Attributes typically correspond to the data that is recorded for an object.

2.3 Strategy: Separation

"Separation refers to distinguishing between a goal or effect and the means or mechanism by which the goal or effect is achieved " [7]. This statement could be translated as follows: Use interfaces or abstract classes as templates or a menu to what functionality is obtainable in the implementation of the interface or abstract class. The template class constitutes the visible part for the implementer and the hidden part is the actual implementation of the desired functionality [8-10]. Obviously, an implementation satisfies an interface or abstract class if the behaviour defined in the interface is provided by the implementation.

Separation entails flexibility through the interchangeable feature that the interface or abstract class offers. The implementation part can be redesigned as long as the implementation satisfies the template.

2.4 Strategy: Composition

"Composition enables building complex system by assembling simpler parts in one of two basic ways; association and aggregation" [11].

The part-whole relationship is crucial to object-oriented programming, as one of the major object-oriented goals is reusability. Composition is one of the most powerful strategies to obtain reusability, but also maintainability [8-11]. Compositions, i.e. interacting parts of an application, could be subject to reuse for a number of different applications. If the interacting parts are carefully generalized, the concept of composition is probably the most powerful strategy [8-11]. "Composition might be viewed as the lego approach to software development, using standardized, specialized parts to construct a wide range of interesting artefacts" [16].

The difference between aggregation and association is the visibility of the fundamental parts or basic functionality. In an aggregation or hierarchical generalization, only the whole, i.e. the aggregated functionality, is visible and accessible. In association the interacting parts are externally visible and may be shared between different object-oriented designs and applications.

2.5 Strategy: Generalization

Generalization is the object-oriented overall abstraction containing hierarchy, polymorphism and patterns [7,10,11]. The commonality amongst these concepts may be of attributes or behaviour of the classes, interfaces or other application components. Generalization identifies possible organization of common properties of abstractions [7,10,11]. Generalization is not abstraction. A generalization is incorporated into the software to simplify the description of entities, such as a part of an application with common properties among the abstractions within the application or namespace.

Polymorphism captures commonality in algorithms or basic functionality, defined in a class or a method. This feature paves the way for reuse and maintainability. Algorithms with a wide scope or a high degree of generalization could therefore be reused and maintained as separate parts of a software project.

Finally; "A pattern expresses a general solution, the key components and relationships, to a commonly occurring design problem. The attributes and behaviour of the individual components are only partially defined to allow the pattern to be interpreted and applied to a wide range of situations "[7, 19]. One of the main reasons that computer science researchers began to recognize patterns was to satisfy the need for good, simple and reusable solutions. These researchers found that it is just a convenient way to of reusing object-oriented code between projects and between programmers. The idea behind design patterns is simple: to catalogue common interactions between objects that programmers have often found useful.

2.6 Summary on object-oriented strategies

The object-oriented strategies, described in Sections 2.1-2.5, form the tools to obtain reusable and maintainable software. Abstraction, separation composition and generalization are widely supported by existing object-oriented languages.

References [6-9] suggest that software reusability and maintainability is strongly related to the use of abstraction, separation, and composition. Furthermore, reusability is, according to [7-9], obtained with inheritance and design patterns through generalization.

3 Reusability and maintainability metrics

A more mathematical and quantitative approach in estimating the degree of reusability and maintainability of an application can be used. One is the measurement of the application level of cyclomatic complexity (CC) [12]. CC has a method focus, whereas abstractness and instability have an object-oriented or namespace focus.

3.1 Cyclomatic complexity

Cyclomatic complexity [12] has been defined [17] as follows:

"CC, introduced by Thomas McCabe in 1976, is the most widely used member of a class of static software metrics. CC may be considered a broad measure of soundness and confidence for a program. CC

measures the number of linearly-independent paths through a program module. This measure provides a single ordinal number that can be compared to the complexity of other programs. It is often used in concert with other software metrics. As one of the more widely-accepted software metrics, it is intended to be independent of language and language format.
The CC of a software module is calculated from a connected graph of the module:

CC = E - N + p
where E = the number of edges of the graph
N = the number of nodes of the graph
p = the number of connected components.

The complexity number is generally considered to provide a stronger measure of a program's structural complexity than is provided by counting lines of code. A large number of programs have been measured, and ranges of complexity have been established that help the software engineer determine a program's inherent risk and stability. The resulting calibrated measure can be used in development, maintenance, and reengineering situations to develop estimates of risk, cost, or program stability. Studies show a correlation between a program's cyclomatic complexity and its error frequency. A low cyclomatic complexity contributes to a program's understandability and indicates it is amenable to modification at lower risk than a more complex program. A common application of cyclomatic complexity is to compare it against a set of threshold values."

Cyclomatic Complexity	**Risk Evaluation**
1-10	A simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk program
greater than 50	untestable program (very high risk)

Table 3.1 *Cyclomatic complexity threshold values.[17]*

Table 3.1 shows some example thresholds for CC, with corresponding risk assessments. Cyclomatic complexity focuses mainly on method maintainability and readability.

3.2 Abstractness and Instability

Quantitative metrics like the level of abstractness and instability have an object-oriented or namespace focus [13]. From quantities like Afferent Coupling (Ca) - the number of classes located outside a namespace that depend on classes located within the analyzed namespace - and Efferent Coupling (Ce) - the total number of classes inside a namespace that depend on classes located outside the analyzed namespace – an estimation of the namespace Instability (I) can be calculated [13].

$I = (Ce \div (Ca+Ce))$.

This metric's range is {0,1}. I=0 indicates maximum stability of a namespace. I=1 indicates maximum instability of a namespace [13].

Abstractness is the total number of interfaces and abstract classes divided by the total number of classes and interfaces of the namespace [13].

Fortunately, there exist numerous standalone and plugin (for integrated development environments) programs which automatically calculate these metrics. In this study, a plugin called dotEasy™ [20] was used.

4. Design Patterns

The authors of [7] have clearly found a need to develop some practical guidelines to achieve the underlying objectives for object-oriented programming. The authors of these guidelines are often referred to as the 'Gang-of-Four', and the guidelines consist of 23 different design patterns. Individual design patterns (DP's) are always identified with a pattern name, the problem it solves, the solution in terms of the collaborating elements that make up the design and the consequences or trade-offs of applying the pattern. In addition to these attributes and descriptions, an UML-diagram is often added to the documentation.

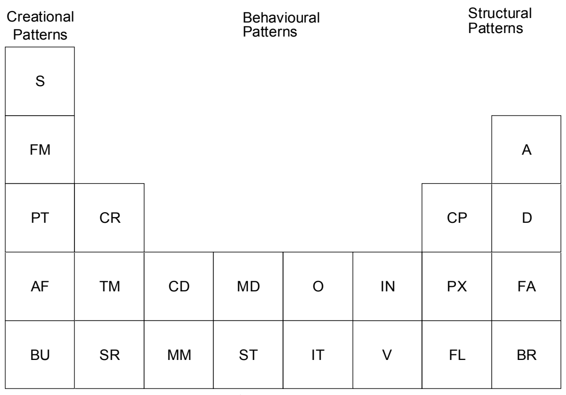
Gamma et al. are by far not the only actors in constructing DP's. BluePrint™ describes 80 and [14] defines a total of 128 separate DP's. [15] adds the application of five software patterns to the development of the collaboration diagrams. This author's GRASP (General Responsibility Assignment Software Patterns) patterns " .. are essentially the key guidelines for assigning responsibilities to classes for fulfilling the system contracts. The patterns show the motivation for determining which objects should create other objects, build associations and modify attributes - the assertions contained in the contracts - by sending messages to the other objects " [18].

4.1 Design Patterns, a definition

A DP can be defined in several ways. One way of defining a pattern is that, to be qualified as a pattern, the underlying problem must occur with some frequency in a given problem domain [7]. The scope of a DP naturally varies with the problem domain, but rare problems should not constitute the base of developing a DP. Furthermore, a DP should incorporate several classes or object and describe the collaboration between them [7,8,9].

4.2 Organizing the Design Patterns

The 23 DP's can be organized using two criteria [7]: the purpose, e.g. a reflection of what the pattern does, and the scope that defines whether the pattern primarily applies to classes or objects. The purpose could be one of the following: creational patterns, structural patterns and behavioural patterns. Creational patterns concern the process of object creation, whereas structural patterns deal with the composition of classes or objects. Finally; behavioural patterns characterize the ways in which classes or objects interact and distribute responsibility. Another way to categorize the patterns, is to use an analogy (see Figure 4.2) to the periodic system:



Creational Patterns			Behavioural Patterns			Structural Patterns	
S							
FM							A
PT	CR					CP	D
AF	TM	CD	MD	O	IN	PX	FA
BU	SR	MM	ST	IT	V	FL	BR

Figure 4.2 *DP periodic 'system'*

The following presents the patterns defined in [7] by describing the individual intent of each pattern. In this section, only the five patterns that I have used in construction the text field parser are described. The description of the eighteen remaining patterns can be found in Appendix A.
This is how [7] describes the purpose of the five patterns I have chosen.

‘Creational patterns:
S, Singleton: Ensures that a class only has one instance, and provide a global point of access to it.
FM, Factory method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
BU, Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Behavioural pattern:
O, Observer: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Structural pattern:
CP, Composite: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.’

5. The toolkit component

Reference [7] defines three broad classes of software and the associated concerns that each class represents. The classes are application programs, toolkits and frameworks.

A toolkit has to be useful in many applications and offer some general-purpose functionality within an application. The value of a toolkit could be measured by the resources saved, e.g. programming personnel and total development time, avoiding recoding frequently used functionality. In this thesis the toolkit is developed as a component to be used in environments such as stand-alone applications and distributed client-server applications.

This thesis demonstrates five of the DP’s that [7] describes and are applicable in solving problem in a toolkit domain.

5.1 Component functionality

My aim was to design a text field parser component in the programming language C#, using five different design patterns. The component should be able to validate user input such as name, addresses and Swedish social security numbers (SSN). This kind of validation is, for example, crucial in database applications, in order to maintain high quality data.

These examples include general validation algorithms of numbers, letters and separators, i.e. blank spaces and minus signs, but also specific algorithms such as calculating control digits and checksums in a SSN.

Building the component, I have used the following assumptions or requirements:

A name can only contain letters. Surnames and given names are separated with a blank space. A valid address is a string that consists of a number of all-letter substrings, separated with blank spaces, and could be closed with a street number. The street number can only contain one letter at the end of the address string. A SSN consist of a valid birth date and four digits, delimited with a minus sign. The last digit is a control digit. Any accidental extra blank spaces should be removed and the first letter in a substring should be transformed to upper case. If a text field does not pass the validation process, an understandable error message should be provided and all events that would normally occur after the validating event should be suppressed.

These requirements gave rise to twelve separate algorithms bundled in 8 classes, constructed using JSP, and will be referred to as the atomic classes. (See Appendix B).

6. Evaluation of the toolkit component

As mentioned in Section 2, software reusability and maintainability is strongly related to the use of abstraction, separation, generalization and composition. These characteristics are therefore worth aiming at, in order to achieve reusable and maintainable software. The design patterns described in section 3 seem to fulfil these aspirations, separately or in combination with other design patterns.
One qualitative way to evaluate the reusability and maintainability of the evolving toolkit component is therefore to study its characteristics in terms of abstraction, separation, generalization and composition. Several qualitative evaluation techniques are possible, e.g. through surveys amongst software developers etc. I have chosen to base my evaluation on the previously-mentioned four qualities instead. To obtain a reasonable level of objectivity in estimating these characteristics, source code, UML-diagrams and so forth are presented in Appendix C (UML-diagrams) and on the accompanying CD-media.

Furthermore, I have incorporated five quantitative methods as an evaluation technique, using measurements such as cyclomatic complexity, afferent couplings, efferent coupling, instability and abstractness.

The following case scenarios, outlined in Sections 6.1 through 6.6, will have a short comment on which object-oriented strategy - see Section 2 - that was used and a table containing the quantitative metrics described in Section 3.

6.1 The worst case scenario

In this scenario, all functionality and algorithms are placed in the C# Form class. The application can be described with this UML-diagram:

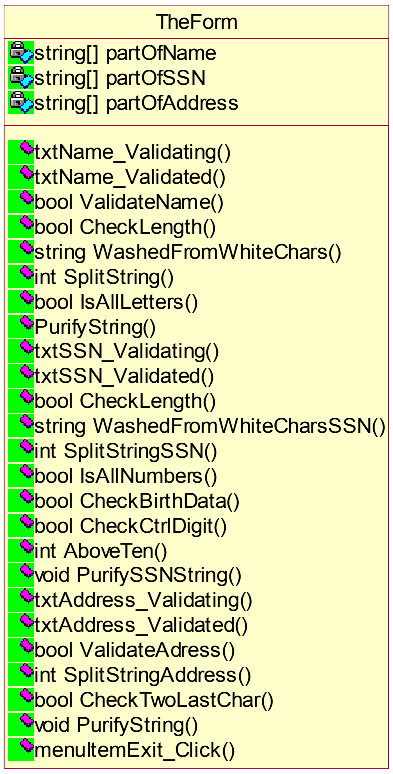


Figure 6.1 *The Form class*

This application contains 802 lines of code and consists of a large number of methods with functionality of a similar kind. For example, the variation in SplitString, SplitStringSSN and SplitStringAddress consist of the use of different string delimiters and the numbers of resulting substrings. If new functionality is demanded, e.g. a text field that validates VAT-registration number, even more methods have to be added into the Form class. This client is destined to be heavy and cumbersome to maintain or reuse. To unload this client, all methods could be placed in a single class.

This application is designed, using an object-oriented programming language, but the design technique is mainly procedural.

One of the reasons for this unsatisfactory condition is the absence of abstractions. For example, an address field validation algorithm make use of ten different methods, a name validation process uses four and so on. The demanded abstractions are possibly ones that encapsulates each validation procedure in predefined classes with individual attributes and behaviours.

6.1.1 The worst case scenario metrics

	Worstcase
Cyclomatic complexity	80
Abstractness (A)	0
Afferent Couplings (Ca)	0
Efferent Coupling (Ce)	0
Resulting Instability (I)	0

Table 6.1.1 *Worst case metrics*

6.2 The use of the Factory method

The Factory method defines an interface for creating an object and delegates the decision on which class is to be instantiated to the subclasses. This implies the existence of subclasses. This forces the programmer to create subclasses like this:

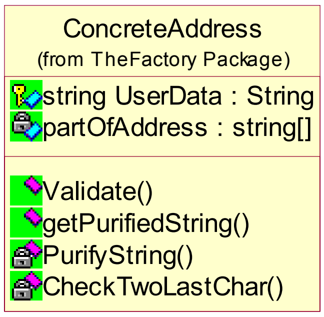


Figure 6.2.(1) *The ConcreteAdress class*

This is an abstraction of the Address field entity, but the Factory method also stipulates separatio*n* in creating objects through an interface.

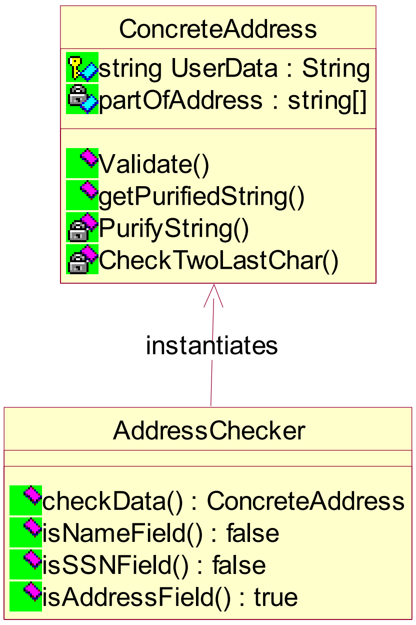


Figure 6.2.(1) *Instantiation of ConcreteAdress*

All the validation algorithms, in this example, are placed in a subclass named StringChecker that contains ten different methods and produces an adequate error message. The remaining three, of the thirteen methods, is found in the concrete classes of ConcreteAddress, ConcreteName and ConcreteSSN. This application suffers to some extent a layer of abstraction. Each atomic validation class (Appendix B) instantiates the error message producing class ReturnValues. In the next section, all the atomic classes inherit the class ReturnValues.

6.2.1 The Factory method metrics

	Factory
Cyclomatic complexity	107
Abstractness (A)	3,5
Afferent Couplings (Ca)	4
Efferent Coupling (Ce)	19
Resulting Instability (I)	1

Table 6.2.1 *Factory metrics*

6.3 The use of the Factory method and Builder Design Patterns

The builder design pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. The abstraction needed is the class ComponentBuilder. ComponentBuilder is the parent of all atomic validation classes (Appendix B) and ComponentBuilder instantiates the class ReturnValues.

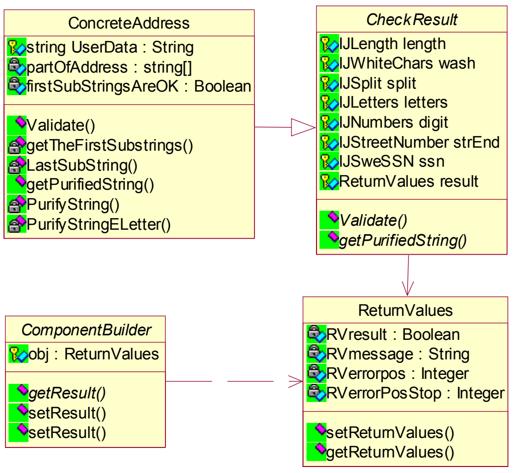


Figure 6.3 *Builder effects*

This application has one definitive shortcoming. The class CheckResult instantiates a fixed number of validating classes, regardless of what kind of validation is needed. It is obvious that yet another abstraction is needed.

6.3.1 The Factory/Builder metrics

	F/Builder
Cyclomatic complexity	146
Abstractness (A)	12,5
Afferent Couplings (Ca)	4
Efferent Coupling (Ce)	19
Resulting Instability (I)	1

Table 6.3.1 *Factory/Builder metrics*

6.4 The use of the Factory/Builder and Composite Design Patterns

The Composite design pattern lets clients treat individual objects and compositions of objects uniformly. In this case the class CheckResult instantiates objects without regard of what kind of validation is needed. The answer is yet another abstraction and a composition of related classes and methods.

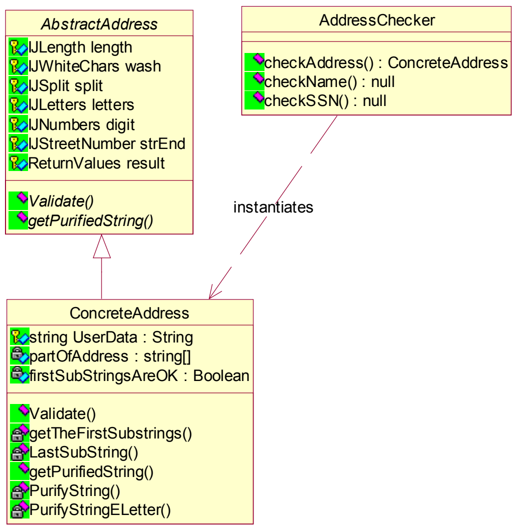


Figure 6.4 *Composite effects*

In this scenario the class ConcreteAddress only inherits and instantiates the validation classes needed for address verification. The feature is obtained by using composition**.** The remaining problem is that the class ReturnValues is instantiated in each time a text field validation check is performed. This superfluous creation of objects is in itself not a problem as C# has an effective garbage collector, but the problem is easy to overcome, see Section 6.5. Such a removal makes it possible to add behavioural separation and features to the application.

6.4.1 The Factory/Builder/Composite metrics

	FB/Composite
Cyclomatic complexity	148
Abstractness (A)	10,75
Afferent Couplings (Ca)	6
Efferent Coupling (Ce)	21
Resulting Instability (I)	1

Table 6.4.1 *Factory/Builder/Composite metrics*

6.5 The use of the Factory/Builder/ Composite/Observer and Singleton Design Patterns

The Singleton design pattern ensures that a class only has one instance and provide a global point of access to it. The Observer design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The first design pattern, Singleton, takes care of the problem mentioned in Section 6.4. In this case the Singleton pattern is applied to the ReturnValues class. The second design pattern, Observer, is also incorporated in the

ReturnValues class. Whenever an error is detected in the validation process, ReturnValues notifies the client, i.e. the input form itself.

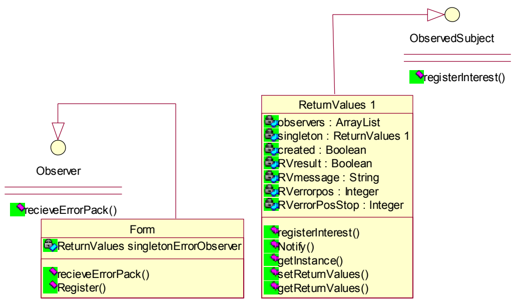


Figure 6.3 *Singleton/Observer implementation*

6.5.1 The F/B/C/S/O metrics

	FBC/SingObs
Cyclomatic complexity	156
Abstractness (A)	4,75
Afferent Couplings (Ca)	7
Efferent Coupling (Ce)	24
Resulting Instability (I)	1

Table 6.5.1 *F/B/C/S/O metrics*

6.6 The effect of hierarchical generalization

It is possible to identify separate elements among the different entities in these applications, e.g. TheFactory, TheBuilder and TheComposite functionality that could constitute the base in forming packages, i.e. facilitate hierarchical generalization (section 2.4). A complete description on package/application architecture is found in Appendix C and the C# source code is enclosed on CD-media. The resulting metrics, after the application of hierarchical generalization, are shown in the following four tables.

List of metrics abbreviations:
CC = Cyclomatic Complexity
A = Abstractness
Ca = Afferent Couplings
Ce = Efferent Coupling
I = Instability

List of application abbreviations:
F = FactoryExample
F/B = FactoryBuilderExample
F/B/C = FactoryBuilderCompositeExample
F/B/C/S/O= FactoryBuilderCompositeSingletonObserverExample
The right hand side of this abbreviations list correspond to the C# project names.

List of application component abbreviations :
CF = Client Form
F pack = Factory package
SC pack = StringChecker package
B pack = Builder package
C pack = Composite package

Worst case	CF
CC	80
A	0
Ca	0
Ce	0
I	0

Table 6.6.[1] *Worst case metrics*

F	CF	F pack	SC pack
CC	20	62	51
A	0	3	0
Ca	0	0	0
Ce	0	0	0
I	0	0	0

Table 6.6.[2] *Factory metrics*

F/B	CF	F pack	B pack
CC	20	62	64
A	0	3	9
Ca	0	0	0
Ce	0	0	0
I	0	0	0

Table 6.6.[3] *Factory/Builder metrics*

F/B/C	CF	F pack	B pack	C pack
CC	20	16	64	49
A	0	3	9	1,3
Ca	0	0	0	0
Ce	0	0	0	0
I	0	0	0	0

Table 6.6.[4] *Factory/Builder/Composite metrics*

F/B/C/S/O	CF	F pack	B pack	C pack
CC	22	16	71	48
A	0	3	3,3	1
Ca	0	0	0	0
Ce	0	0	0	0
I	0	0	0	0

Table 6.6.[5] *F/B/C/S/O metrics*

6.7 Implementing new functionality

Lets assume that I want to add a phone number text field to the client form in the F/B/C/S/O-application (section 6.5) and validate the user input. The requirements for this text field could look something like this. A phone number always has four area code digits and five subscriber digits. The two strings are, for example, delimited with a slash character. The atomic classes (section 5.1) that could be used are IJLength, IJNumbers, IJSplit and IJWhiteChars.

The first step is to design an abstract class called AbstractPhoneNumber that instantiates these atomic classes. Then a concrete class called ConcretePhoneNumber is constructed, that implements AbstractPhoneNumber. The last phase consists of the creation of a PhoneNumberChecker that implements FormChecker. The latter then returns an object of ConcretePhoneNumber-type.

In three steps, new functionality has been added to the application.

7. Results

The five DP's that I have used utilize the object-oriented strategies mentioned in Section 2, i.e. abstraction, separation, composition and generalization. In this perspective, DP as a concept promoted maintainability and reusability.

Before using hierarchical generalization, the resulting metrics where discouraging. The cyclomatic complexity was as high as 156 (Table 6.5.1), nota bene, the application was untestable and burdened with very high risk (Table 3.1). The afferent and efferent couplings were so high, that the resulting instability reached its maximum. The application named “Worst case” proved to have the best metric results (Table 6.1.1). These results indicated that the worst-case scenario application was the most reusable and maintainable program.

After introducing hierarchical generalization, the individual metrics improved in the DP examples. For example, the instability metrics went down to zero and the individual readings on cyclomatic complexity improved from being untestable to moderate or high risk. The best test results where to be found in using the Factory method (Table 6.6.[2]), Factory/Builder (Table 6.6.[3]) and Factory/Builder/Composite (Table 6.6.[4]).

The sharp drop in cyclomatic complexity, regarding the Factory package, between the F/B example and F/B/C example was a result of incorporating the Composite package. The latter was given the responsibility to manage the abstract and concrete classes of the text field classes.

The increase of cyclomatic complexity, regarding both the Client Form and the Builder package, between F/B/C example and F/B/C/S/O was assignable to the insertion of singleton and observer capabilities.

The Builder package, with the highest readings on cyclomatic complexity, was clearly the bottleneck in all applicable arrangements. This package contained the atomic classes mentioned in Section 5.1.

An interesting result was that the sum of the individual readings – the row sum - on cyclomatic complexity after the use hierarchical generalization, was comparable to the ones before this rearrangement.

Finally Section 6.7 described a three-step procedure on how to incorporate new functionality to the F/B/C/S/O-application.

8. Result Analysis

The results in the previous section could be encumbered with a variety of errors. One source of error could be that I have chosen the wrong design patterns, or that I have used wrong object-oriented strategies. The results could also be misleading because of ineffective design of algorithms, especially regarding the Builder package with its high reading on cyclomatic complexity.

One might argue that too many design patterns have been put to use in a single application such as the F/B/C/S/O-application (section 6.5). The opposite view, that further abstraction, separation, composition or generalization is required, could also be valid in order to reduce the cyclomatic complexity.

9. Conclusion

In Section 6.7 I described how an extended functionality in form of phone number validation functionality could be implemented in the software, using three basic design steps. Let us assume that the metrics analysis had prevailed and the ‘worst case’ application from Section 6.1 where chosen as the solid ground for maintainability and reusability. In this case the programmer had to decide which overloaded method to use and combine these methods into a concrete phone number validating method. As the “Worst Case” application contains 802 lines of code, this could constitute a somewhat distressing situation for the programmer.

To me it seems easier to maintain and reuse, for example, the F/B/C/S/O-application. This application has an architecture or a level of abstraction that forms a template on how new functionality could be incorporated and how the collaborating parts are put together. Through inspection of UML-diagrams or even source code, this template quality of the application becomes clear.

The use of design patterns or object-oriented strategies promotes the creation of templates or understandable patterns like the ones in Sections 6.1-6.7. I suggest that these patterns are more easily understood by humans, as this feature addresses basic human associative and abstract capabilities. In other words, a programmer ought to find it easier to follow a template, rather then finding a sequential or procedural solution to a complex problem.

A common situation is that individual programmers have to interact with colleagues in order to fulfil a commission work. To be able to create an effortless communication between practitioners, these individuals have to *share* a conceptual language for communicating insight and experience about the problems and their solutions. Furthermore, forming a common pattern language for conveying the structures and mechanisms of our architectures allows us to intelligibly reason about them. In my opinion, design patterns have the ability to constitute the base for such a mutual language.

In order to express this opinion, the poor metric result (Table 6.6[5]) of the Builder package must be explained. The Builder package contains all the atomic classes that form the very base of the application. It can be regarded as the application repository of validation functionality. To achieve a lower level of cyclomatic complexity, it is possible to organize these independent atomic classes in different sub packages, regarding the different scope of the individual classes. For example, sub-packages like General Text Functionality, General Digit Functionality etc. could be created. In my private opinion, a division like this would have cluttered the overall architecture and thereby reduced the application maintainability and reusability.

10. Discussion

The creation of design patterns has become an industry. I have seen design patterns for file creation, Null object pattern, Lock object pattern, Demilitarized pattern etc. It would have been great to be able to use a total of 128 design patterns [14], in an effortless manner, but at this time of writing it seems a bit overwhelming. For my one use, I think about seven patterns are enough to cope with the applications I am involved in.

On a personal level, having about three years of experience of object-oriented programming, the concept of design patterns has brought a deeper insight in the ideas and mechanisms behind object-oriented programming.

Adversaries to design patterns often argue that the whole concept of design patterns limits the programmer's creativity. For my part, I consider design patterns as a solid and needed architectural tool that constitutes the borders of a creativity play ground. Programming is and should be fun, but on the other hand something useful must come out.

The use of metrics in evaluating software was a complete new experience for me. Used in a proper manner, I truly believe that metrics can be of substantial use designing software. The need for hierarchical generalization and the identification of the troublesome Builder package was a direct result of the use of metrics.

11. Future Work

The Standish Group [1] conducts recurrent studies on the underlying factors for IT-projects success or failure. In these studies features like project management tools, practices and methodologies, application development principles and processes, project cost, project success evaluations, team development and management are studied. It would be interesting if research institutes like the Standish group incorporated studies regarding the possible correlation between the use of a common language such as DP and the resulting effects on budget overruns and delay of delivery.

12. Acknowledgements

First and foremost I would like to thank Dr. Steven Kirk for his unflagging support of this thesis. I also would like to thank Dr. Ludwik Kuzniarz, who introduced me to world of Design Patterns. Last but certainly not least, I thank my wife and my beloved daughter Sofia. Without their love, support and understanding, this thesis would never have been written.

13. References

[1] The Standish Group (1995). “The CHAOS Report”[www-document],.URL http://www.pm2go.com/sample_research/chaos_1994_1.asp

[2] C. Böhm and G. Jacopini.” Flow diagrams, turing machines and languages with only two formation rules”. Communications of the ACM, 9(5):366--371, 1966.

[3] Edsger, W. Dijkstra. “A Case against the GO TO Statement”. ACM 11(3): 147–148, 1968.

[4] Jackson, M. A. “Principles of program design”. Academic Press, 1975

[5] Booch, G, Rumbaugh, J, Jacobson, I. “The unified Modelling Language. Addison-Wesley, 1999

[6] Skansholm, J. ”Java direkt”. Studentlitteratur, 2001

[7] Gamma, E. Helm, R. Johnson, R Vlissides, J. ”Design Patterns”. Addison-Wesley, 1995.

[8] Weisfeld, M. McCarty, B. ”The Object-Oriented Thought Process”. Sams, 2000

[9] Meyer, B. ”Object-Oriented Software Construction”. Prentice Hall, 2000

[10] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. ”Object-Oriented Modelling and Design”. Prentice Hall, 1991

[11] Booch, G. ”Object-oriented Analysis and Design”. Addison-Wesley, 1994

[12] McCabe, T., Watson, A. "Software Complexity." Crosstalk, Journal of Defense Software Engineering 7, 12 (December 1994): 5-9.

[13] Martin, R. ”Agile Software Development, Principles, Patterns, and Practices”. Prentice Hall, 2002

[14] Grand, M. ”Patterns in Java”. John Wiley & Sons, 1998

[15] Larman, C. ”Applying UML Patterns”. Prentice Hall, 2002

[16] http://people.cs.vt.edu/~kafura/cs2704/composition.html

[17] http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html

[18] http://www.objectsbydesign.com/books/applying_uml.html

[19] http://ei.cs.vt.edu/~kafura/java/Chapter1/generalization.html

[20] www.doteasy.addr.com

Appendix A

The complete list of design patterns according to Gamma, E. Helm, R. Johnson, R Vlissides, J.

1. Creational patterns:

PT, Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
AF, Abstract factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

2. Behavioural pattern:

CR, Chain of responsibility: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
CD, Command: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
IN, Interpreter: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
IT, Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
MD, Mediator: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
MM, Memento: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

3. Structural patterns:

A, Adapter: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
BR, Bridge: Decouple an abstraction from its implementation so that the two can vary independently.
D, Decorator: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.
FA, Façade: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
FL, Flyweight: Use sharing to support large numbers of fine-grained objects efficiently.
PX, Proxy: Provide a surrogate or placeholder for another object to control access to it.
ST, State: Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.
SR, Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
TM, Template method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
V, Visitor: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Appendix B

The Atomic Classes

General functionality classes

Class: IJLength

Method:

```
public void CheckLength(string testStr,int testLength,bool checkType)
```

Usage:

Checks if `testStr.Length` is bigger `than testLength when checkType` is false.
Checks if `testStr.Length` is exactly `testLength when checkType` is true.

Method:

```
public void CheckLength(string testStr)
```

Usage:

Checks if `testStr.Length` is bigger `than testLength when checkType` is false.
Checks if `testStr.Length` is exactly `testLength when checkType` is true.

Class: IJNumbers

Method:

```
public void IsAllNumbers(char[] x)
```

Usage:

Evaluates if `char[] x` only contains numbers. If `x[i]` is not a number, the error position `i` is sent to the class ReturnValues.

Method:

```
public void IsAllNumbers(int lengthFirstSubstring, char[] x)
```

Usage:

Evaluates if `char[] x` only contains numbers, when the total string contains substrings. If `x[i]` is not a number, the error position `i` is sent to the class ReturnValues.

Class: IJWhiteChars

Method:

```
public string WashedFromWhiteChars(char[] x, bool noWhites)
```

Usage:

Removes all white spaces in a substring, e.g. 6 5 is transformed to 65 or
“O   K” becomes “OK”

```
public string WashedFromWhiteChars(char[] x)
```

Usage:

Repeating white spaces, between substrings, is substituted to a single white space.

Class: IJLetters

Method:

```
public void IsAllLetters(char[] x)
```

Usage:

Evaluates if `char[] x` only contains letters. If `x[i]` is not a letter, the error position `i` is sent to the class ReturnValues.

Class: IJDate

Method:

`public void CheckDate(string yyyymmdd)`

Usage:

Evaluates if `string yyyymmdd` is a valid date

Class: IJSplit

Method:

`public string[] SplitString(string inData, string strDelimit)`

Usage:

Split strings according to a string delimiter. A bit easier than the built-in version 'Split'.

Method:

`public void SplitStringOK(string inData, string strDelimit, int nrOfSubStrings)`

Usage:

A bit like IndexOf (checks if a char exist in a string), but SplitStringOK evaluates a number of occurrences of a string delimiter in the tested string. Example: Does the string “65-07 99” contain three (`nrOfSubStrings`)substrings delimited by a minus sign (`strDelimit`).

Specific functionality classes

Class: IJSweSSN

Method:

`public void CheckCtrlDigit(char[] theSSN)`

Usage:

Evaluates a checksum for a candidate Swedish social security number. Calls CheckDate.

Class: IJStreetNumber

Method:

`public void StreetNumber(string lastAddressPart,int lengthOfFirstAddressPart)`

Usage:

Evaluates three scenarios and if an error occurs, the error position `i` is sent to the class ReturnValues.

1. A street number can not begin with a letter, e.g. f 5
2. There are to many letters in this substring to represent a street number, e.g. 5 ff
3. The last substring contains rubbish, i.e. §5#

Appendix C

UML-diagrams

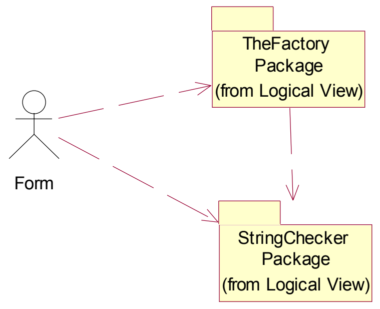

TheFactory Package (from Logical View)
Form
StringChecker Package (from Logical View)

Figure C.1 *Factory example. Package overview*

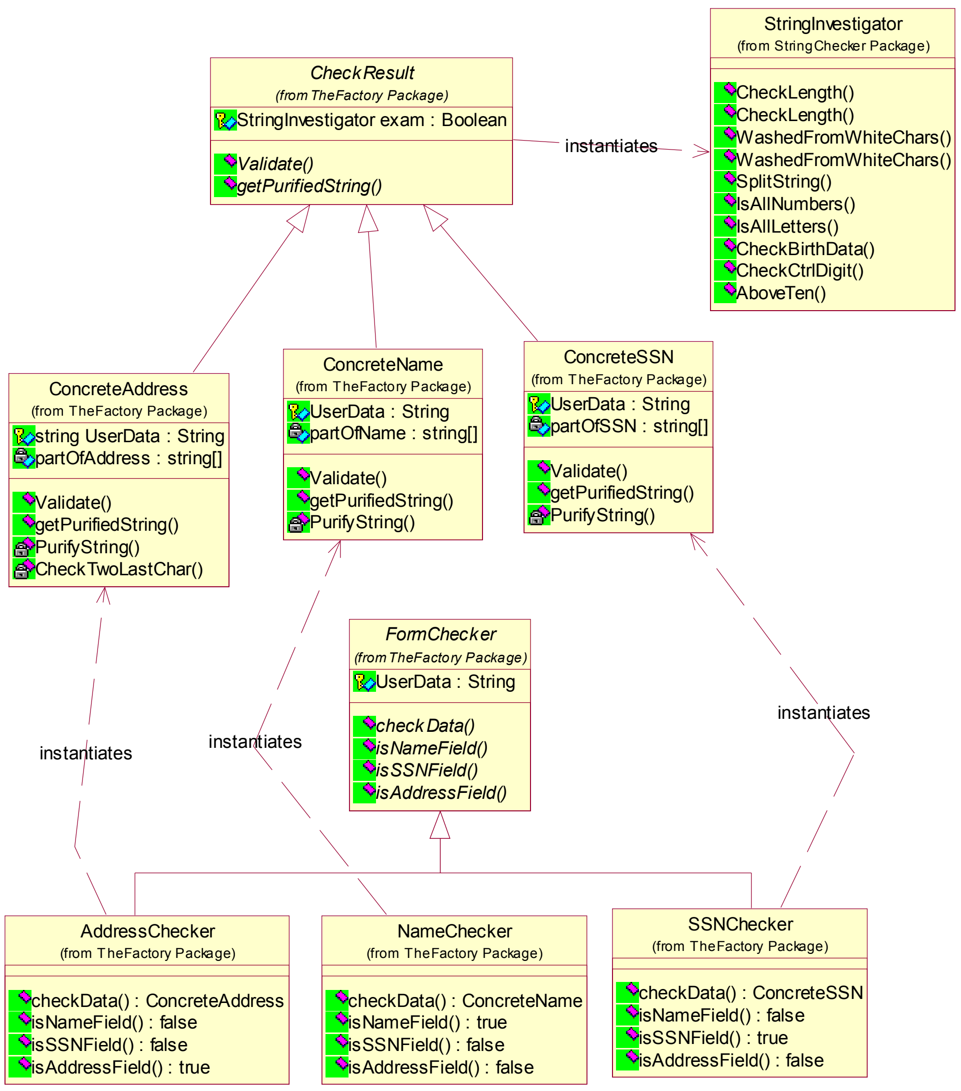


Figure C.2 *Factory example. Factory and StringChecker packages*

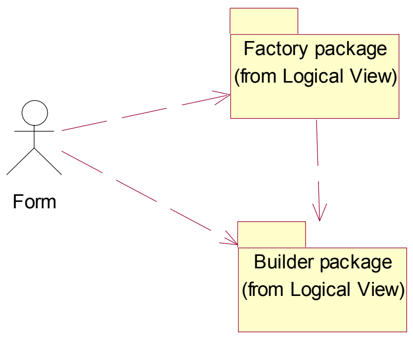


Figure C.3 *Factory/Builder example. Package overview*

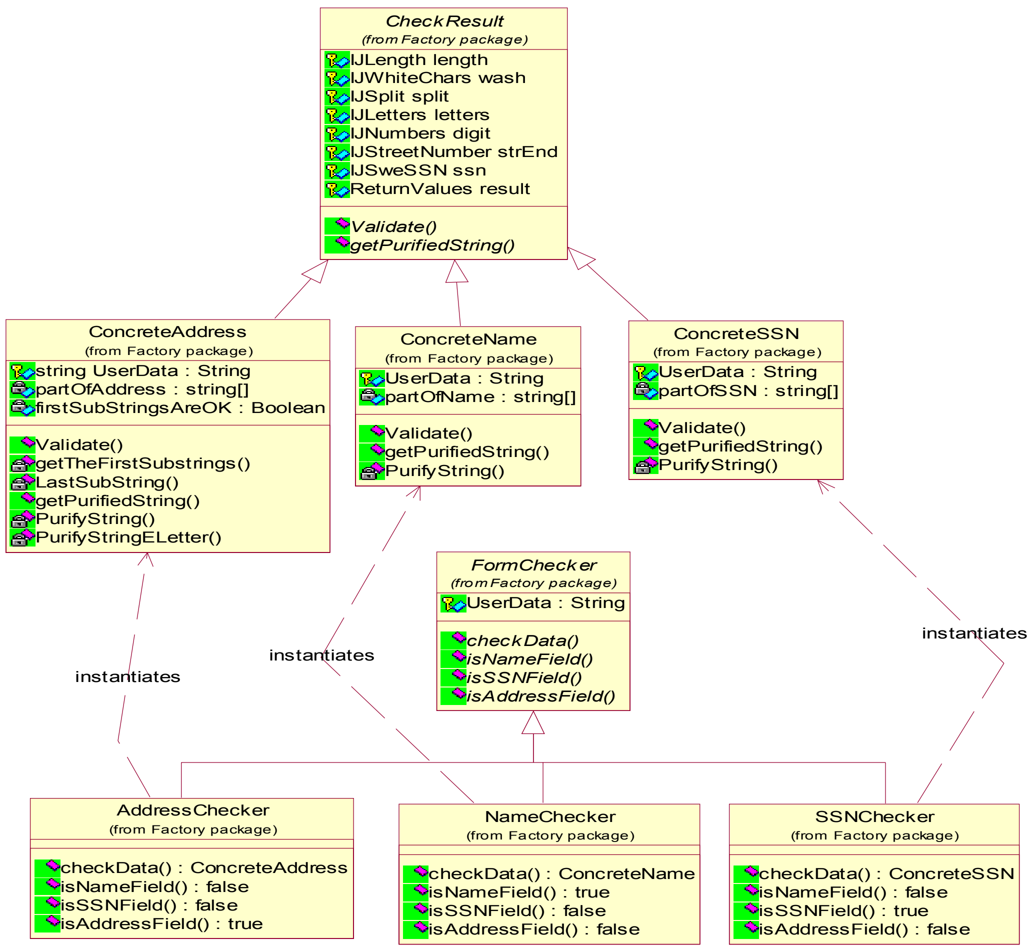


Figure C.4 *Factory/Builder example. Factory package.*

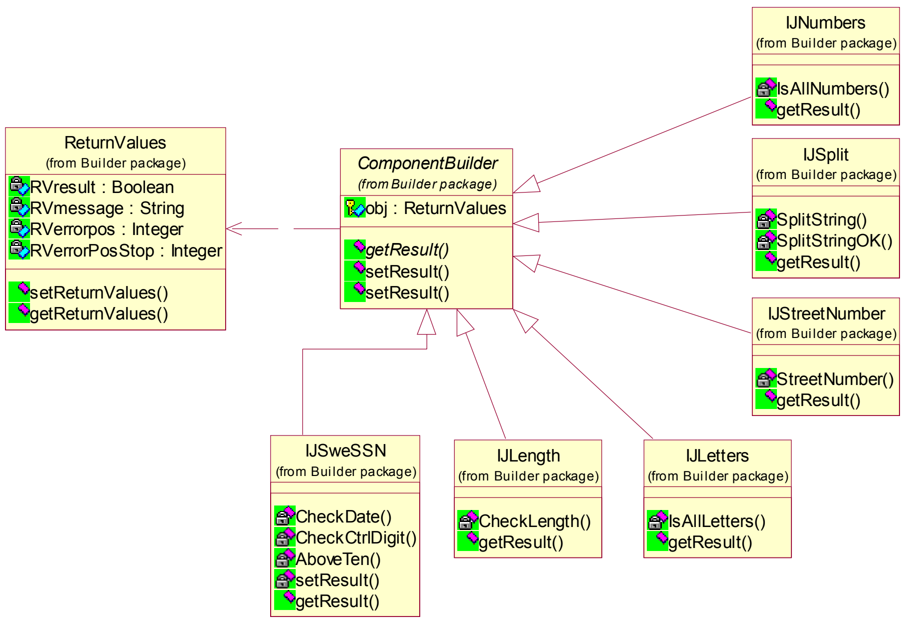


Figure C.5 *Factory/Builder example. Builder package.*

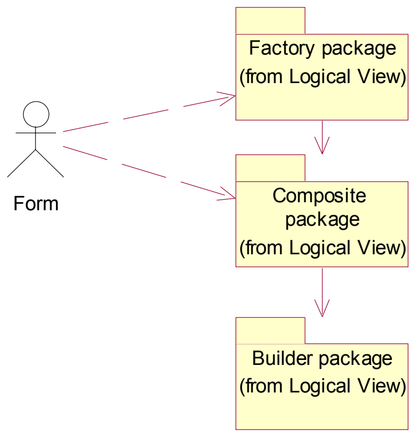


Figure C.6 *Factory/Builder/Composite example. Package overview.*

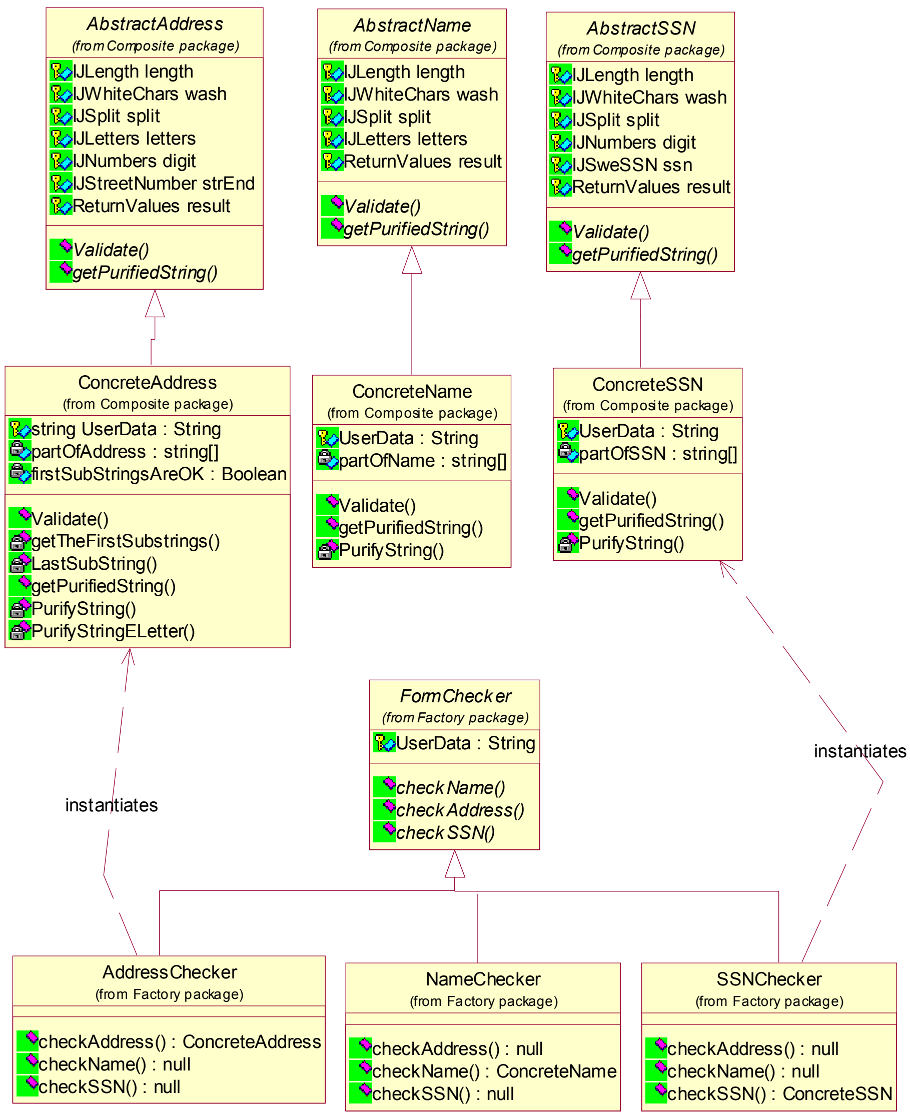


Figure C.7 *Factory/Builder/Composite example. Factory and Composite packages.*

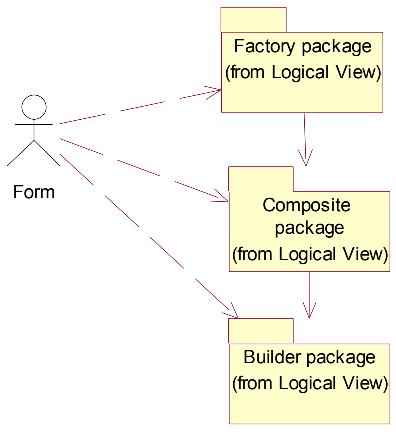


Figure C.8 *Factory/Builder/Composite/Singleton/Observer example. Package overview.*

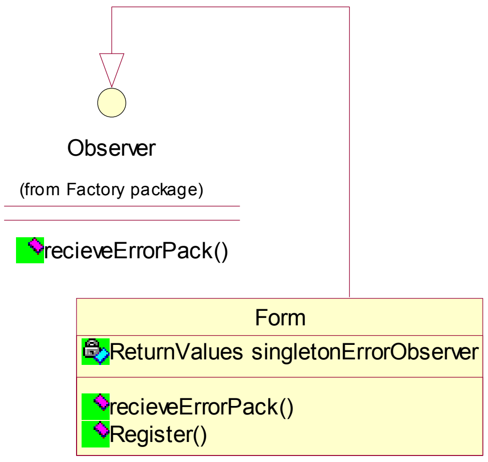


Figure C.9 *Factory/Builder/Composite/Singleton/Observer example. The Form class.*

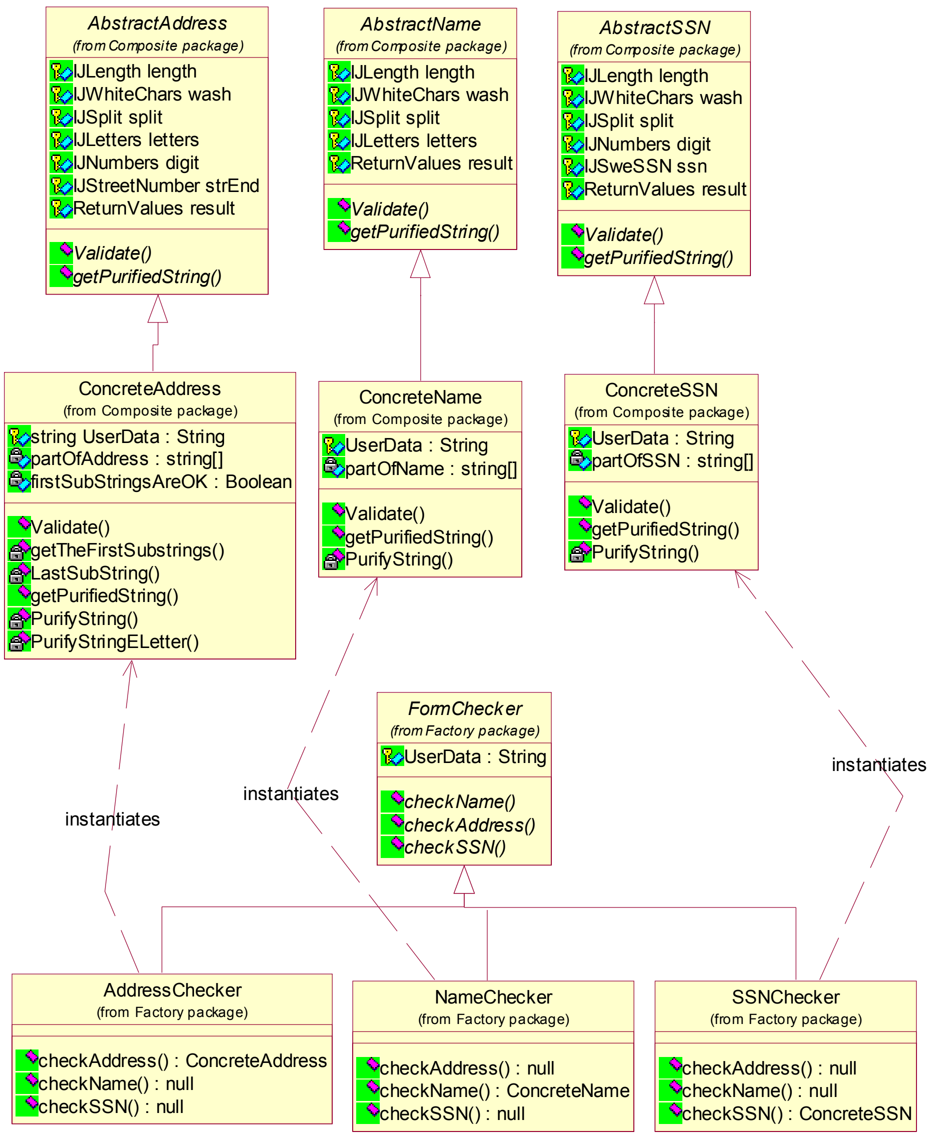


Figure C.10 *Factory/Builder/Composite/Singleton/Observer example. Factory and Composite packages.*

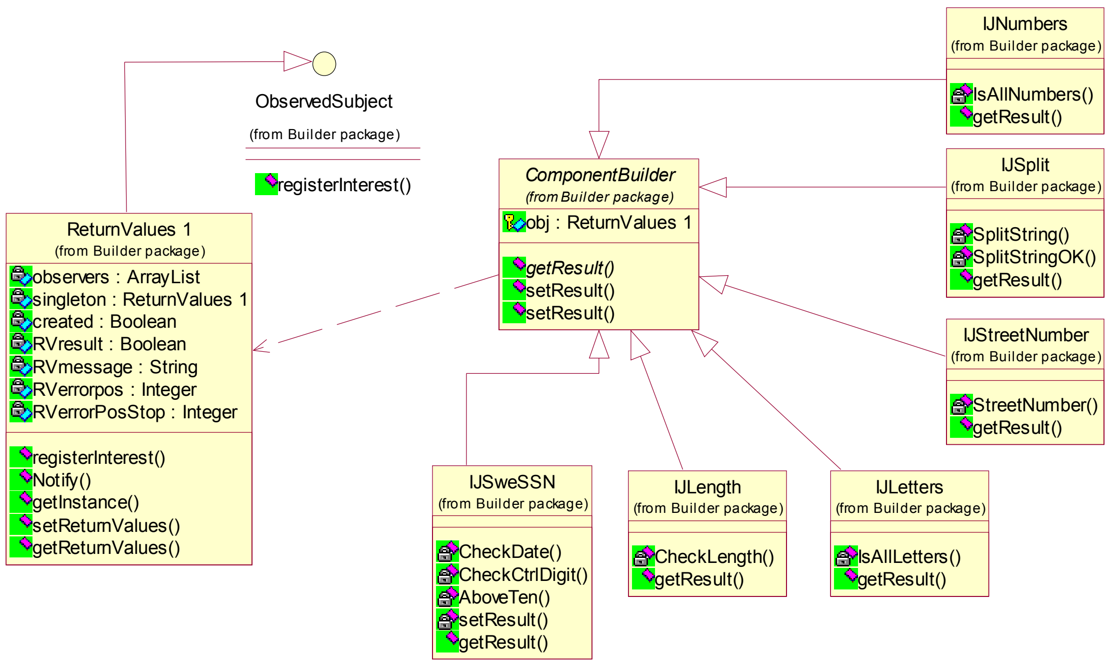


Figure C.11 *Factory/Builder/Composite/Singleton/Observer example. Builder package with Observer and Singleton capabilities.*