Correct by Construction

Author:
J. Maree FOURIE

March 19, 2020

"Writing is hard; writing is hard because thinking is hard. It is easier to think you are thinking."

Leslie Lamport

Abstract

Correct by Construction

by J. Maree FOURIE

ToDo. ...

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

CbyC	**C**orrect **by** **C**onstruction
TLA	**T**emporal **L**ogic **A**ctions

Chapter 1

Process and Technique

Software development can be seen as being composed of two parts:

- process;
- technique.

Process describes the people involved in developing the software and the interactions between them. The process can describe things like team structures, work planning, reviewing work done and delivering the software.

Technique describes technical aspects of software development. Technique can describe things like what programming languages and technologies to use, how the software is structured and tested, and also how the software is delivered.

Process and technique can overlap and inform each other as in the case of software deployment. If we wanted to do continuous delivery we would have to review the completed software on an as-needed basis and not a fixed schedule (process). We might then also structure the software as micro-services and not a monolithic application (technique).

Some development methodologies focus more on the process and others on technique.

Agile software development (Agile) Agile focuses on delivering useful software in a timely manner by focusing on process. Agile's only opinion on technique is that the design should be kept simple.

Correct by Construction (CbyC) CbyC uses formal methods as a technique along with a process that aligns with the formal methods. Formal methods are mathematical rigorous techniques used to validate software designs and code.

If we combine Agile process with the CbyC technique and process it may result in a methodology that is both rigorous and productive.

Chapter 2

Correct by Construction (CbyC)

2.1 Overview

The CbyC software development methodology proposes to limit software defects by [4]:

1. making it difficult to introduce errors, and
2. detecting and removing errors as early as possible.

CbyC technique emphasises the use of mathematically rigorous tools. CbyC proposes using mathematically formal language to define the specification and high-level design of a system. This provides a precise description of the system's behaviour and a precise model of its characteristics. Using a mathematical language also allows the use of automated tools to verify the specification and design [4].

CbyC code is designed around information flow. The information flow is defined using a contract-based notation. The contract-based notation is used to define the abstract state of the program and the information relationships across boundaries. CbyC proposes using programming languages and tools that allow for the code and contracts to be validated using static analysis [4].

2.2 Correct by Construction Process

The CbyC process is described in terms of the artefacts it produces (Figure 2.1):

- System requirements specification;
- Formal specification;
- Formal design;
- System test specification;
- INFORMED design;
- Code;

Using the CbyC process every process artefact can be validated. The semantic gap between process artefacts are kept small. This makes it possible to show that later process artefacts conform to earlier process artefacts [3].

Here follows a description of each artefacts and how it is created.

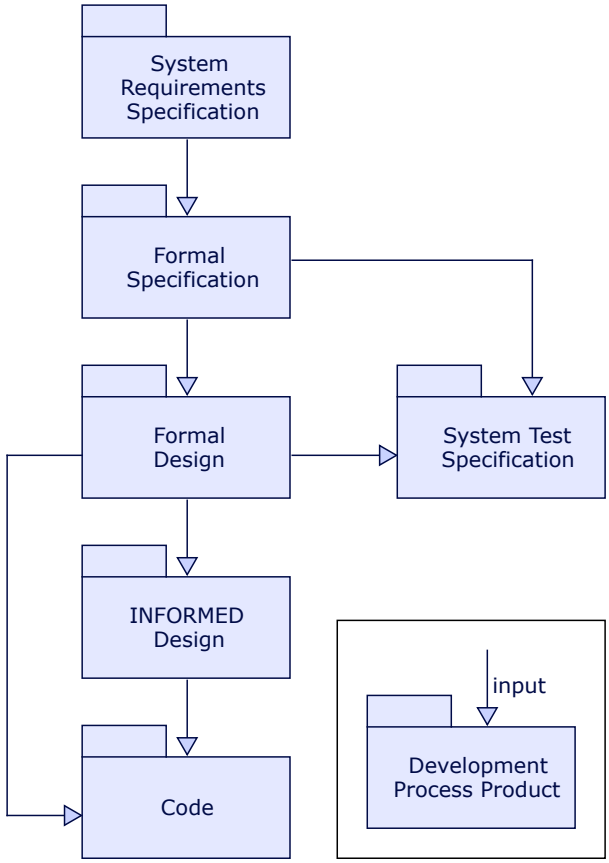

FIGURE 2.1: CbyC process artefacts.

2.2.1 System Requirements Specification

Requirements analysis identifies the needs of the stakeholders, the desired behaviour of the system, and any non-behavioural system characteristics [3].

Requirements management extends throughout the system's development, but it is most significant at the beginning, where it is used to identify [3]:

- the stakeholders (who have an interest in the development and use of the system);
- the system boundary (to clarify the scope of the project and the interfaces to external systems);
- the expected use (in terms of interactions between users and the system);
- system properties (such as security properties, performance properties, etc).

Each requirement should be traceable through every process artefact [3]. The requirements are compiled into the System Requirements Specification document.

The System Requirements Specification is created to [3]:

- early in the project clarify the system's boundary (what is in scope and what is out of scope, and the necessary interfaces to external systems);
- agree on the system requirements with all of the stakeholders;

- document the requirements with enough procession so that the Formal Specification can be developed with minimal customer input;
- clarify and document the assumptions about the behaviour of the environment external to the system;
- identify and manage conflicting expectations between stakeholders.

2.2.2 Formal Specification

The Formal Specification unambiguously describes what the system will do. This helps the developer and client gain a common understanding of the system [3].

The level of abstraction is important. The Formal Specification should not describe the system's implementation. Internal details are deliberately left abstract. Interactions with the external environment are specified, but may also be abstract [3].

The Formal Specification is written in mathematical notation with an English narrative. The specification is divided into small components that can be reasoned about individually. The small components are then combined to describe the system as a whole [3].

The Formal Specification is created because it [3]:

- provides an unambiguous description of what the system does. This is important for client approval of the system's behaviour;
- can be shown to be complete;
- can be formally verified and proven consistent.

2.2.3 Formal Design

The Formal Design elaborates on the abstract aspects of the Formal Specification to explain how the system will be implemented. The Formal Design describes the system in terms of concrete state and operations using types that are easily implementable. The Formal Design is the source of the required functional behaviour used during implementation [3].

The Formal Design is written in the same mathematical notation as the Formal Specification. This means that the Formal Design can be formally verified and errors can be uncovered before implementation starts [3].

The Formal Design is created because it [3]:

- provides an unambiguous description of how the system will accomplish what the Formal Specification requires;
- can be shown to be complete;
- can be formally verified and proven consistent.

2.2.4 System Test Specification

The systems tests described in the System Test Specification has to show that the system implements the behaviours described in the Formal Specification. System tests are more complete than acceptance tests. Acceptance tests only have to show that the system functions as described in the System Requirements Specification. System Testing aims to achieve 100% coverage of the Formal Specification so that all behaviours described in the Formal Specification are executed at least once [3].

The Formal Design is a refinement of the Formal Specification. Therefore the Test Specification can also be written against the Formal Design if you want to tests details of the design [3].

All system tests are specified in the System Test Specification before being executed. The tests are specified as scenarios that can occur during typical system usage. The System Test Specification documents the expected outcome of the test. Each system test is linked to functionality described in the Formal Specification/Design[3].

Where code coverage metrics are needed, it can be done during the System testing. This allows us to question the use of any code that cannot be covered by a system test [3].

System tests are specified because [3]:

- system tests focuses on testing the behaviour of the whole system against the expected (specified) behaviour;
- system tests find faults caused by interaction of the system's modules;
- system tests complement static analysis by confirming the system's dynamic behaviour.

2.2.5 INFORMED Design

CbyC design methodology is based on information flow. The information flow is defined using contract-based notation. The contract-based notation is used to define the abstract state of the program and the information relationships across boundaries [4].

The **IN**formation **F**low **OR**iented **ME**thod of object **D**esign (INFORMED) design provided an architectural framework in which to perform the implementation. Consideration of information flow at the design stage results in programs with the desirable properties of abstraction, encapsulation, high cohesion and loose coupling [3].

The INFORMED design aids maintenance and upgrades of the software by providing a route-map from the Formal Design to the code [3].

An INFORMED Design is created because it [3]:

- focuses on the system architecture and ensures that the architecture fits the information flow model;
- provides the mapping from the Formal Design to the Code before writing the code.
- complements the Formal Design without duplicating functional information.

2.2.6 Code

We start by writing the module specification. The specification is the public interface and the contracts governing the global state of the module. The contacts specify how inputs are allowed to influence outputs [3].

After writing the specification we implement the module body. The Formal Design is detailed enough to make the mapping from the design to the code simple [3].

Modules providing infrastructure are developed early. Modules are implemented in an order that allows system functionality to be added incrementally. This means that a basic system can be built as soon as possible and functionality is added in subsequent builds. This has the advantage of addressing code integration risks as early as possible [3].

By using languages that allows static analysis we can evaluate the code contracts at build time, if the code builds, it is correct as specified by the contracts [3].

Chapter 3

Agile using Scrum

3.1 Overview

The Manifesto for Agile Software Development sets out the overarching principles of agile software development [1]:

> "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - **Individuals and interactions** *over* processes and tools
> - **Working software** *over* comprehensive documentation
> - **Customer collaboration** *over* contract negotiation
> - **Responding to change** *over* following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more."

This is further expanded on by the Twelve Principles of Agile Software [1]:

> "We follow these principles:
>
> 1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
> 2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
> 3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
> 4. Business people and developers must work together daily throughout the project.
> 5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
> 6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
> 7. Working software is the primary measure of progress.
> 8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
> 9. Continuous attention to technical excellence and good design enhances agility.

10. Simplicity–the art of maximizing the amount of work not done–is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly."

Scrum is an agile project management methodology. It focuses on short feedback loops. Scrum is described in terms of roles, events, and artefacts. The Scrum process workflow is shown in Figure 3.1.

3.2 Scrum roles

The scrum team consists of a Product Owner, the Development Team, and a Scrum Master. The scrum team is self-organizing and cross-functional. The Scrum team has all the skills necessary to complete assigned tasks without outside assistance [13].

3.2.1 Product Owner

The Product Owner is one person. The Product Owner may represent a committee of people, but the Product Owner must only be one person. The Product Owner is responsible for maximizing the value of the product resulting from the work done by the Development Team. The product owner is the only one responsible for managing the Product Backlog. The rest of the team may help manage the backlog but the Product Owner is the only accountable person [13].

3.2.2 Development Team

The Development Team is a small team consisting of between three and nine professional people. The Development Team members do the work necessary to deliver a potentially releasable increment of the product. A completed potentially releasable increment of the product is required at the Sprint Review [13].

3.2.3 Scrum Master

The Scrum Master is the servant-leader of the Scrum Team. The Scrum Master facilitates the interaction between outsiders and the team. The Scrum Master steers the interaction between team members to maximize the value create by the Scrum Team [13].

3.3 Scrum Events

Scrum Events are used the create a regular rhythm and minimizes the need for irregular disrupting causing meetings [13].

3.3.1 Sprint

The Sprint is the core of Scrum. A Sprint is a period, no longer than a month, in which a usable and potentially releasable increment of the product is created. The length of the Sprint does not change throughout the development process. The next Sprint starts directly after the previous Sprint finishes. The Sprint contains and consists of the Sprint Planning, Daily Scrum, the development work, the Sprint Review, and the Sprint Retrospective [13].

3.3.2 Daily Scrum

The Daily Scrum is a 15 minute event for the Development Team. During the Daily Scrum the Development Team plans the work for the next 24 hours that is needed to achieve the Sprint Goal [13].

3.3.3 Sprint Review

A Sprint Review is held at the end of the Sprint to inspect the Product Increment and adapt the Product Backlog [13].

3.3.4 Sprint Retrospective

The Sprint Retrospective is held at the end of the Sprint to allow the Scrum Team an opportunity for introspection to plan for improvements that can be done in the next Sprint [13].

3.4 Scrum Artefacts

Scrum Artifacts represent work done or provide information about the development process in order to provide transparency [13].

3.4.1 Product Backlog

The Product Backlog is an ordered list of every known requirement of the product. The Product Backlog is always evolving as new requirement are discovered. It is the only source of required changes to the product. The Product owner is responsible for the Product Backlog, this includes its content, availability, and ordering [13].

3.4.2 Sprint Backlog

The Sprint Backlog is the set of Product Backlog items selected for the Sprint and a plan for delivering the product Increment, realizing the Sprint Goal [13].

3.4.3 Increment

The Increment is the sum of all the Product Backlog items completed during a Sprint and the value of the increments of all previous Sprints [13].

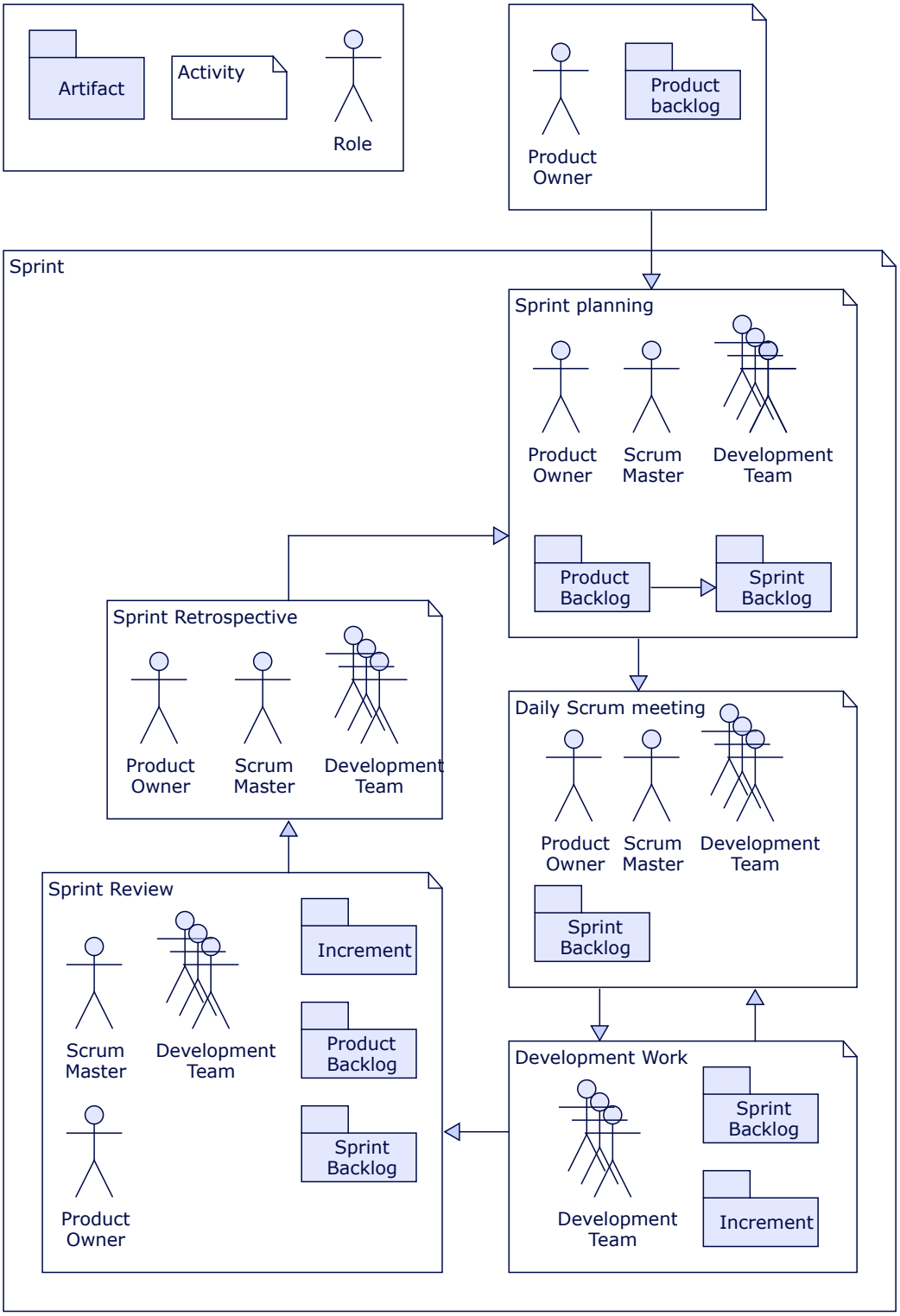

FIGURE 3.1: The Scrum workflow.

Chapter 4

Using Correct by Construction with Agile Scrum

4.1 Why not just Scrum

The driving force behind Agile Scrum is the ability to react to changing requirements. To evaluate requested changes the changes have to be implemented. If we evaluate this against the Agile manifesto:

Responding to change: In order to evaluate a change: a story has to be created, and the story has to be refined and sized. Then, depending on the priority, existing work has to be moved to make resources available to implement the change.

Working software: We have to implement the requested change in order to validate it. If the requested change does not work we might end up with invalid or non-functional software. The changes then has to be rolled back.

Customer collaboration For a change to be actioned the team will reason through it on a whiteboard. This reasoning is based on opinion and results in arguments that cannot be resolved without implementing the change.

4.2 Augment Scrum with Correct by Construction

The CbyC process is designed to validate changes at every step. By augmenting Scrum with CbyC we reduce the need to implement a change before it can be validated. Thereby streamlining the development process and reducing the risk of having broken or invalid software. Figure 4.1 shows how the Scrum process can be augmented with CbyC.

4.2.1 Roles

Team Architect

We add the Team Architect to help the Product Owner create the Formal Specification. This role can be shared amongst the team members and does not have to belong to a specific person.

4.2.2 Events

Product Planning

In Scrum the Product Owner can create the Product Backlog without assistance from the rest of the team. The Product Owner can still create the System Requirements Specification on

his own, but the Formal Specification is written in a mathematical language and the Product Owner might need help writing it. To facilitate the writing of the System Requirements Specification and the Formal Specification we add the Product Planning event. During the Product Planning event the Team Architect helps the Product Owner write the Formal Specification. Once a change has been added to the Formal Specification and verified, user stories can be added to the Product Backlog. Requirements should be traceable through the System Requirements Specification and Formal Specification to the story in the Product Backlog.

Sprint

To the normal Sprint development work we add, from CbyC, the Formal Specification, Formal Design, System Test Specification, and the INFORMED Design. The stories in the Sprint Backlog tells us what work we are going to do to create the increment. We take the work described by the story and the specification in the Formal Specification and we then accordingly update the Formal Design, System Test Specification and INFORMED design. After verification we write the code to create/update the Increment.

4.2.3 Artefacts

We add the artifacts as describe by CbyC:

1. System Requirements;
2. Formal Specification;
3. Formal Design;
4. System Test Specification;
5. INFORMED Design.

Increment

We create the Increment as described by CbyC Code Artifact.

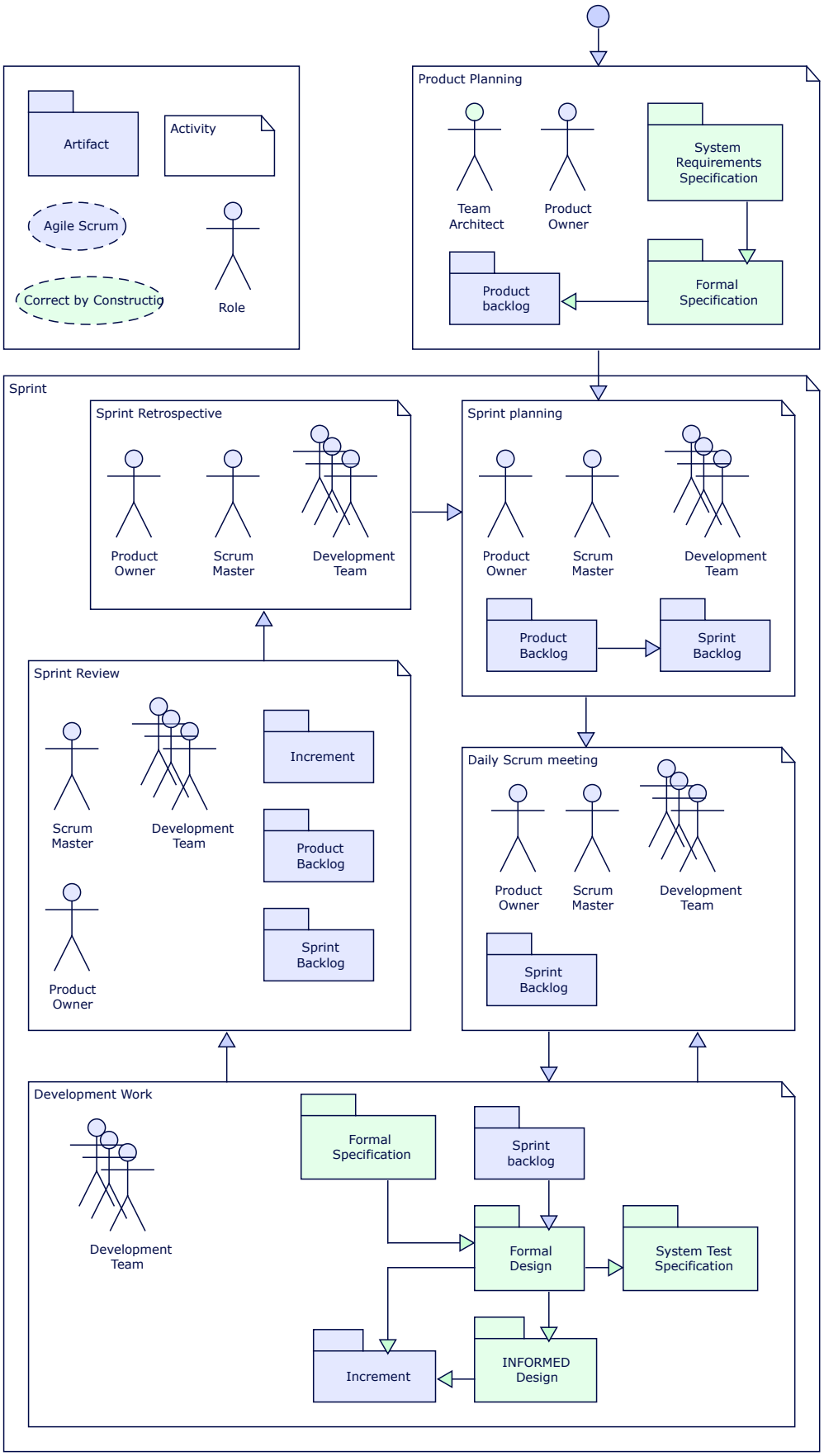

FIGURE 4.1: The Scrum workflow augmented with CbyC.

Chapter 5

Correct by Construction Agile Scrum Technique

We have now described the process we are going to follow. In this chapter we describing the technique we are going to use the create the artifacts.

To create all the artifacts we need a mathematical specification language and a programming language.

5.1 Mathematical Specification Language (TLA$^+$)

The Formal Specification and Formal Design has to be written in a formal mathematical language. CbyC suggests using Z [6]. I hound that Z has very little tool support and an inactive community. I decided to use TLA$^+$. TLA$^+$ has frequently updated tooling and an active community. To get started with TLA$^+$ visit the "Learning TLA$^+$" website [9].

A TLA$^+$ specification is a boolean expression defining all the valid states the system my occupy. The TLA$^+$ sttol

5.2 Programming Language (C#)

CbyC suggests using the Ada SPARK programming language because it has native support for code contracts and can verify the code using static analysis [6]. Very little line of business software is built using Ada. To make this more relevant to the my work I decided to use C#. The problem with C# is that is has no native support for code contracts and also does not have sufficient static analysis tools capable of verifying code correct. This means that we will have to find workarounds for these short comings.

5.2.1 Code Contracts

CbyC program design is based on information flow expressed as code contracts [4]. C# does not have native code contracts any more [11], but it is very simple to implement code contracts using standard C# language features.

Code contracts are an application of Hoare logic. Using Hoare logic we can show that a program correctly implements a specification if, given a set of preconditions derived from the specification, the program never violates a set of post conditions derived from the specification. We use the Hoare triple to represent the contract [8]:

$$\{P\}Q\{R\}$$

where:

P are the preconditions;

Q is the program;

R are the postconditions.

With some helper methods we can implement code contracts using standard C# language features. We are modelling our code contracts on the Design by Contract design [12].

Design by Contract has two specification levels:

- method preconditions and postconditions;
- class invariants.

Preconditions are a set of conditions that state what the method requires from the client (code calling the method) in order to function correctly.

Postconditions are a set of conditions that state what the method will ensure happens at the end of it's execution.

LISTING 5.1: Writing code contracts using helper methods and classes.

```
public int Method(...)
{
    //Preconditions
    Require(/*Consition*/, "Message")
        .Require(/*Consition*/, "Message");

    /// Method Code ...

    //Postconditions
    Ensure(/*Consition*/, "Message")
        .Ensure(/*Consition*/, "Message");

    return result;
}
```

Class invents are a set of conditions that will always be true after all executions.

LISTING 5.2: Writing code contracts using helper methods and classes.

```
public class Account
{
    ...

    public int Method(...)
    {
        /// Method Code ...
        ClassInvariant();
        return result;
    }

    private void ClassInvariant()
    {
```

```
            Invariant(/*Consition*/, "Message")
                .Invariant(/*Consition*/, "Message");
        }
}
```

5.2.2 Verification

Testing is usually the main method of verification and validation. Unit testing is inefficient because unit testing is ineffective and expensive. Unit testing is ineffective because most errors are interface errors, not internal errors in units. Unit testing is expensive because you have to build test harnesses to test units in isolation [6].

Instead of unit testing code to show correctness we just want to verify that the contracts are not violated by the code are valid. When we know the code never violates the contracts we only need to create tests that show the different parts of the system function together.

We can represent a program or section of a program as a mathematical function with all possible legal permutations of inputs seen as the function's domain and the output of the program the function's range.

It would be impractical to write unit tests that cover the complete input domain of the program. Instead we write tests that take random points on the input domain as input. We are not going to verify the outputs because if the outputs passes the contracts we assume the outputs are valid. This will eliminate the oracle problem [2].

When we tests smaller sections of the program separately random testing provides similar, adequate, code coverage as compared with determinisms testing [2][7].

We will be using FsCheck to create out property based tests. FsCheck is a .net implementation of QuickCheck [5].

Chapter 6

TLA+

We specify a system as a set of possible behaviours representing a correct execution of the system. We define a behaviour as a sequence of states. A state is defined as an assignment of values to variables.

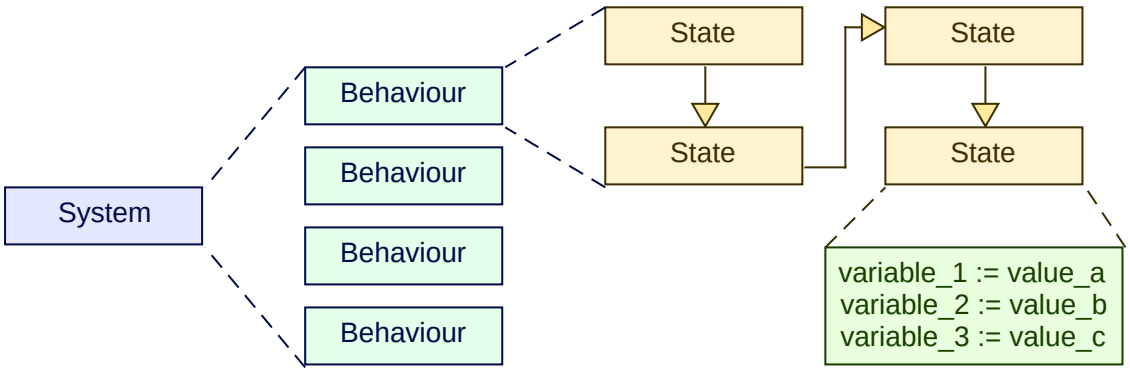

FIGURE 6.1: System specification.

6.1 Writing a Specification

6.1.1 Boilerplate

System are specified in module files. You should use the module name for the file name with a .tla extension.

LISTING 6.1: Writing code contracts using helper methods and classes.

```
---------------------------- MODULE module_name ----------------------------
EXTENDS \* list modules
VARIABLES \* list variables
\* system specification
-----------------------------------------------------------------------------

=============================================================================
```

6.1.2 Variables, Type Invariant, Initial Predicate

Decide what variables you are going to use to describe your system. It is usually better to use high-level, abstract descriptions of the system's data structures in a specification.

TLA+ only has four basic kinds of values: strings, integers, booleans, and model values[]. To create more intresting types we

6.1.3 Next-State Action

First, pick the variables and define the type invariant and initial predicate.

6.1.4 Temporal specification

6.2 Info

state function An ordinary expression (one with no prime or □) that can contain variables and constants.

state predicate A Boolean-valued state function.

safety properties Specify valid behaviour. Specify what is allowed to happen. If a safety property is violated, it is violated at a specific point in the behaviour. If a safety property has been satisfied it means the safety property has not been violated by any step in the behaviour so far.

liveness properties Specify what behaviour that cannot be violated at any particular instant.

Formally a temporal formula F assigns a Boolean value, which we write $\sigma \vDash F$, to a behaviour σ. We say that F is true of σ, or that σ satisfies F, iff $\sigma \vDash F$ equals TRUE.

1. A state predicate, viewed as a temporal formula, is true of a behaviour iff it is true in the first state of the behaviour.
2. A formula $\Box P$, where P is a state predicate, is true of a behaviour iff P is true in every state of the behaviour.
3. A formula $\Box[N]_v$, where N is an action and v is a state function, is true of a behaviour iff every successive pair of steps in the behaviour is a $[N]_v$ step.

$\Diamond$ **F** It asserts that F is not always false, which means that F is true at some time. We usually read $\Diamond$ as eventually, taking eventually to include now.

$F \rightsquigarrow G$ asserts that whenever F is true, G is eventually true. We read $\rightsquigarrow$ as leads to.

$\Diamond\langle A\rangle_v$ asserts that eventually an $\langle A\rangle_v$ step occurs

$\Box\Diamond F$ asserts that at all times, F is true then or at some later time. In particular,$\Box\Diamond\langle A\rangle_v$ asserts that infinitely many $\langle A\rangle_v$ steps occur.

$\Diamond\Box F$ asserts that eventually (at some time), F becomes true and remains true thereafter. $\Diamond\Box[N]_v$ asserts that, eventually, every step is a $[N]_v$ step.

6.3 How to structure a document

6.4 basics

6.5 Verify

Chapter 7

Specification

Appendix A

TLA+

A.1 TLA$^+$ Toolbox

Use the TLA$^+$ Toolbox to evaluate your specifications. You can download the latest toolbox from the tlaplus GitHub repository [14].

There is more information about the toolbox on the TLA+ website [10]

A.2 More information

Appendix B

FsCheck

B.1 Installing

B.2 More information

Appendix C

Code Contracts

C.1 Hoare logic

C.1.1 Program execution

$$\{P\}Q\{R\}$$

If the assertion P is true before the program Q executes, then the assertion R will be true after Q has executed.

Axion of Assignment

$$\vdash P_0\{x := f\}P$$

x is a variable identifier;

f is an expression;

P_0 is obtained from P by substituting x with f;

Rule of Consequence

$$\text{If } \vdash \{P\}Q\{R\} \text{ and } \vdash R \supset S \text{ then } \vdash \{P\}Q\{S\}$$
$$\text{If } \vdash \{P\}Q\{R\} \text{ and } \vdash S \supset P \text{ then } \vdash \{S\}Q\{R\}$$

This rule allows the strengthening of the precondition and/or the weakening of the postcondition.

Rule of Composition

$$\text{If } \vdash \{P\}Q_1\{R_1\} \text{ and } \vdash \{R_1\}Q_2\{R\} \text{ then } \vdash \{P\}(Q_1; Q_2)\{R\}$$

Rule of Iteration

$$\text{If } \vdash \{P \wedge B\}S\{P\} \text{ then } \vdash \{P\} \text{ while } B \text{ do } S\{\neg B \wedge P\}$$

C.2 C# code contract

We implement the contract condition tests using the below listed helper methods and classes.

LISTING C.1: Implementing code contracts using helper methods and classes.

```
public static class Contract
{
    public static Requirer Require(bool condition, string message)
    {
        if (!condition)
            throw new ConstactRequireException(message);
        else
            return new Requirer();
    }

    public static Ensurer Ensure(bool condition, string message)
    {
        if (!condition)
            throw new ConstactEnsureException(message);
        else
            return new Ensurer();
    }

    public static Invarianter Invariant(
        bool condition, string message)
    {
        if (!condition)
            throw new ConstactInvariantException(message);
        else
            return new Invarianter();
    }
}

public class Invarianter
{
    public Invarianter Invariant(bool condition, string message)
    {
        if (!condition)
            throw new ConstactEnsureException(message);
        else
            return this;
    }
}

public class Ensurer
{
    public Ensurer Ensure(bool condition, string message)
    {
        if (!condition)
            throw new ConstactEnsureException(message);
        else
            return this;
    }
}
```

```
public class Requirer
{
    public Requirer Require(bool condition, string message)
    {
        if (!condition)
            throw new ConstactRequireException(message);
        else
            return this;
    }
}
```

We can now write contracts using the helper methods and classes.

Require conditions is the first code in the method. If we need to compare pre and post execution values we can create an anonymous object to store the old values. Only store value types and not reference types because the referenced values can change and will not be preserved by the old object.

The class invariant conditions are listed in a method so that they can be executed in all methods.

Ensure and class invariant conditions are the last code in the method before it returns.

LISTING C.2: Writing code contracts using helper methods and classes.

```
public class Account
{
    private int _limit;

    public int Limit
    {
        get => _limit;
        set
        {
            Require(value > 0, "Limit must not be 0.");
            _limit = value;
            ClassInvariant();
        }
    }

    public int Total { get; private set; }

    public int Add(int value, int discount)
    {
        Require(value >= 0, "Value must be positive.")
            .Require(
                discount >= 0 && discount <= 50,
                "Discount must be positive and not more than 50%");
        var old = new { Total };

        // method logic

        Ensure(Total > old.Total, "The total will not decrease.");
```

```
        ClassInvariant();
        return discountedValue;
    }

    private void ClassInvariant()
    {
        Invariant(Total >= 0, "The total will not be negative.")
            .Invariant(Limit > 0, "Limit will not be 0.");
    }
}
```

Bibliography

[1] Kent M. Beck et al. *Manifesto for Agile Software Development*. http://agilemanifesto.org. [Online; accessed 7-December-2019]. 2001.

[2] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP* 46 (Jan. 2000). DOI: 10.1145/1988042.1988046.

[3] David Cooper. *Tokeneer ID Station EAL5 Demonstrator: Summary Report*. https://www.adacore.com/tokeneer. [Online; accessed 12-December-2019]. Aug. 2008.

[4] Martin Croxford and Dr. Roderick Chapman. "Correctness by Construction: A Manifesto for High-Integrity Software". In: *The Journal of Defense Software Engineering* 18.12 (Dec. 2005), pp. 5–8.

[5] *FsCheck*. https://fscheck.github.io/FsCheck/index.html. [Online; accessed 3-December-2019].

[6] Anthony Hall and Roderick Chapman. "Correctness by Construction: Developing a Commercial Secure System". In: *IEEE Software* 19.1 (Jan. 2002), pp. 18–25.

[7] Richard Hamlet. "Random Testing". In: *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[8] C.A.R. Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580.

[9] Leslie Lamport. *Learning TLA+*. https://lamport.azurewebsites.net/tla/learning.html. [Online; accessed 29-January-2020]. Sept. 2019.

[10] Leslie Lamport. *The TLA$^+$ Toolbox*. https://lamport.azurewebsites.net/tla/toolbox.html. [Online; accessed 4-December-2019]. 2019.

[11] Immo Landwerth. *Are Code Contracts going to be supported in .NET Core going forwards?* https://github.com/dotnet/docs/issues/6361. [Online; accessed 06-February-2020]. Aug. 2018.

[12] Bertrand Meyer. *Object-Oriented Software Construction*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997. ISBN: 978-0-13-629155-8.

[13] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. https://www.scrum.org/index.php/resources/scrum-guide. [Online; accessed 27-January-2019]. Nov. 2017.

[14] *TLA$^+$ GitHub release*. https://github.com/tlaplus/tlaplus/releases/latest. [Online; accessed 4-December-2019].