Concept/Topic: *Component Modeling and CRC Cards*

Context

The book that accompanies the lecture, "Effektive Softwarearchitekturen" (Starke (2015)) explains that solution strategy/design is about *structure* and *technology*.[1] This request for structure can be met by identification of candidate components and their continuous refinement (as we begun to do in lessons 4 and 5). Technology concepts are designed/decided via styles, patterns, reference architectures, frameworks/containers, starting with big decisions for instance about layering, Client/Server Cuts (CSCs); component containers; architectural style such as Client/Server (or SOA and microservices, which will be covered in later lessons). See separate fact sheet for more details.

The components that we deal with in solution strategy are candidate components. A candidate component is an intermediate/preliminary architectural element used for planning, decision making, architectural prototyping. The candidate components are subject to continuous refinement and consolidation efforts (architectural refactoring).

Steps and Component Definition by Viewpoint

Four overall component modeling steps (identification, followed by specification, realization, and composition) are shown in the figure "Component Modeling Steps". Each step has its own notations and techniques (to use a term from the method anatomy introduced in lesson 2). For instance, CRC cards can be used for in component specification. Note that the sequential organization of the figure does *not* suggest a waterfall approach or "big design upfront"; it is perfectly fine to iterate (with short cycles).

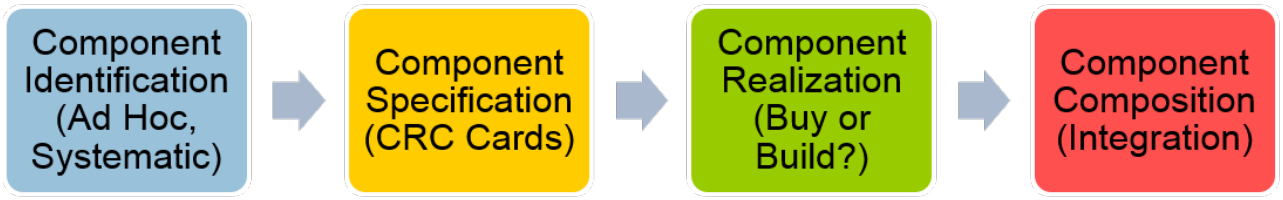

Figure 1: *Component Modeling Steps*

A *candidate component* is an architectural element in the logical viewpoint grouping related responsibilities that jointly satisfy one or more (non-)functional requirements so that design and implementation work can be planned and component realization decisions can be made: Examples in the "T" case study are "Validate Address" and "Create PSTN" (from the architecture overview diagram). Types of candidate components are: channel components, business logic beans, and adapter components (one type per layers). Frequently used notations are informal rich picture (sometimes dubbed "boxology" or even "marchitecture"), C4 component diagram (or stereotyped UML class diagram), and CRC cards.

Implementation components, on the contrary, reside in the development viewpoint. They group one or more classes (assuming that OOP is used) and provide an interface that hides their implementation details e.g. in JEE and Spring. Their dependencies are made explicit and managed; they are versioned. Notations to specify

[1] Structure is then covered in detail in Chapter 4, and many technical concepts in the (long) Chapter 7.

implementation components are UML class or component diagrams, detailed C4 component diagrams, or simply code.

Deployment units then bundles one or more implementation components to be hosted on one or more nodes (physical viewpoint).

Component Identification (CI)

A rule of thumb is to identify one candidate component per layer and feature/entity without knowing too much about them yet (they are merely placeholders making sure nothing gets forgotten later in design and development).

OOAD and business modeling techniques and patterns can help with CI:

- C. Larman's GRASP principles, "UML Components" by J. Cheesman/J. Daniels (Cheesman and Daniels (2000)) and x-Driven Design (x=Responsibility, Domain, Behavior)
- Business modelling discipline in UP/UML (as introduced for instance here[2]) and business process modeling and prioritization by business analyst. A sample notation is BPMN (see lecture on "Wirtschaftsinformatik"). Another technique that can be used is the IBM Component Business Model (CBM) approach.
- Industry reference models can also serve as starting points, e.g. eTOM (telecommunications) and ACORD (insurance).

It is also ok (but can turn into an anti pattern easily) to simply use structures of/in existing systems.

ZIO Heuristic

This is different from the more elaborate, rather UML-centric CI practice suggested by J. Cheesman and J. Daniels in "UML Components" (Cheesman and Daniels (2000)), but inspired by it and compatible.

Input The CI Input includes: FRs, NFRs/QAs, architectural vision statement (expressed in table form as suggested by arc42, or as an X-statement as shown in sample solution to exercise 3) plus analysis-level domain model (e.g. UML class diagram) as taught in SE1/SE2.

Process First iteration: Find Candidate Components (CanCos)

1. Add one *channel component* per primary actor (which can be human users or external system consuming services of the system under construction) shown in the context diagram (white box view of the system under construction)
2. Add one component per layer for each of these (it is ok not to add if already present; add responsibility in that case) :
   - Feature (user story, use case); this might not be needed for all features, e.g. on data access/database layer due to reuse, but for now "if in doubt put it in"
   - Domain model partition (as defined in SE1/SE2, making a preliminary assessment, assumption making and personal judgment are perfectly fine to bring in here; more systematic approaches can be applied and will be discussed elsewhere)
3. Add one *adapter component* per backend system appearing in the context diagram

A channel component is an inbound (upstream) adapter serving all requests from one particular user group over one technology/protocol (such as mobile app, RIA, fat client). An adapter component bundles outbound requests to one particular backend (downstream) system and hides its platform specifics from the higher layers.

[2] https://www.ibm.com/developerworks/rational/library/360.html

This initial CI effort can be followed by a CRC card brainstorming or workshop (to find responsibilities, collaborators) for component refinement (or specification). This activity can start from use case scenarios (see below for more information on CRC cards and related techniques).

Starting in second iteration, *Architectural Refactoring* (of CRC cards) is performed to address quality concerns such as security and management (from QAS). Potential realization technologies (implementation candidates) are listed, and a sanity/completeness check run (possibly supported by reference architectures). Architectural meta issues (a.k.a. recurring architectural decisions) e.g. about security and management) can already be considered now.

Output The CI output is a refined set of requirements and domain model, early C4 diagrams (or basic list of candidate components), and/or CRC cards.

Component Specification (CS)

Responsibility-Driven Design (RDD) as introduced in the book "Object Design" by R: Wirfs-Brock (Wirfs-Brock and McKean (2002)) is a predecessor of several contemporary methods. Among other things, it introduces the notion of roles: "Role stereotypes from Responsibility-Driven Design are a fundamental way of seeing objects' responsibilities. Think of them as purposeful oversimplifications that help you identify the gist of an object's responsibilities. You can use stereotyping early on to characterize your early candidate objects. Later, you can use stereotyping to characterize your design." (source: RDD tutorial[3]).

The six primary role stereotypes in RDD are:

- An *information* holder knows and provides information.
- A *structurer* maintains relationships between objects and information about those relationships
- A *service provider* performs work and in general offers services
- A *controller* makes decisions and closely directs others' actions
- A *coordinator* reacts to events by delegating tasks to others
- An *interfacer* transforms information and requests between distinct parts of a system. User interfacers translate requests from the user to the system (and vice versa). External interfacers usually "wrap" other system APIs. There are other interfacers in complex systems that serve as the "front door" to subsystems.

The channel and adapter components from above are examples of interfacers.

CRC Cards

A notation that is well suited for CS is the CRC Card format. CRC stands for Components, Responsibilities, Collaborators (CRC) here. The original CRC cards were invented by W. Cunningham, who also invented Wikis and Technical Debt metaphor, and published in his OOPSLA 1989 paper[4]. They were popularized in RDD (for programming and object design level). On the architectural level, they are used in the POSA 1 book (Buschmann et al. (1996)). See a blog post by M. Stal[5] for more background and motivation.

An annotated template (with teaser questions) is shown in the figure "CRC Card Notation Explained".

The figure "CRC Card Notation Example" gives an example of a filled out card (an entire layer is described here, rather than a single component in that layer).

One usage scenario for CRC cards is as a workshop or "design thinking" element, sometimes involving a role-playing game: Each participant takes the role of an object/a candidate component (one CRC card). A ball

[3] http://www.wirfs-brock.com/PDFs/A_Brief-Tour-of-RDD.pdf
[4] http://c2.com/doc/oopsla89/paper.html
[5] http://stal.blogspot.ch/2006/12/architects-toolset-crc-cards.html

Component: **[Name of Component (with optional Role Stereotype)]**	
Responsibilities: • [What is this component capable of doing (services provided)?] • [Which data does it deal with?] • [How does it do this in terms of key system qualities?]	*Collaborations (Interfaces):* • [Who invokes this component (service consumers)?] • [Whom does this component call to fulfill its responsibilities (service providers)?] • [Any external connections (both active and passive)?]
Candidate implementations technologies (and known uses): • [Which technologies, products (commercial, open source) and internal assets realize the outlined component functionality (responsibilities)?]	

Figure 2: *CRC Card Notation Explained*

Component: **Business Logic Layer (RDD Role: Controller)**	
Responsibilities: • Receives service requests • Authenticates request originator, authorizes service request • Keeps track of processing state (workflow, not page flow!), manages transaction boundaries • Accesses business objects • Performs calculations • Updates business objects and application state • Sends response • Logs layer activities in audit trail	*Collaborations (Interfaces):* • Presentation layer • Data access layer • Cross-cutting infrastructure features (e.g., security subsystem) • Component container (Inversion of Control)
Candidate implementations technologies (and known uses): • Service layer, process layer in “T” case study • BLL in PQG case study • domain.model package in DDD Sample Application • HSR enterprise applications such as unterricht.ch, AVT • …	

Figure 3: *CRC Card Notation Example*

representing control flow is thrown around to specify data flow, (a)synchrony, etc. The resulting collaborations and responsibilities are recorded on CRC card. Another usage scenario is specification and documentation tool. Finally, the cards can support decision making during component realization, starting with the "buy vs. build" decision: use software package, commercial middleware, or open source software? Can an operating system feature fullfill the responsibilities (such as crontab as a job scheduler)?

Component Dynamics and Component Interaction Diagrams (CIDs)

Architecture is not only about structure, but also about flow. Sunny days are different from cloudy or rainy days (high load, error cases) from a control flow point of view (see Environment entry in QAS template in SMART NFR Elicitation fact sheet). The rationale why component dynamics need to be investigated can be structured according to the FURPS taxonomy (from lesson 2):

- User stories/use cases require multiple steps to be performed when black box (system context) us refined into white box (logical component model)
- Reliability might require load balancing and hot standby, which in turn requires components such as watchdog and circuit breakers that have to communicate with each other and with those implementing the user stories/use cases.
- Performance is hard to judge when only looking at static structure,
- Security attacks come as internal or external stimuli, responses require system-internal activity that has to be designed and evaluated

Component dynamics are modeled in the component specification phase and, if needed, during component realization. Columns are candidate or implementation components (depending on purpose of diagram/model); interactions might originate from CRC card (or another early design artifact). Valid notations for CIDs are:

- UML 2.5 interaction diagrams (with components as stereotyped classes): sequence, collaboration/communication diagram; activity diagram, state machine (note that UML is available via some Web services as well, e.g., Plant UML and Web sequence diagrams)
- C4 Dynamic diagram
- Informal lists of steps

Both normal operations and error situations should be investigated when designing component dynamics.

CIDs allow reasoning about load and miscellaneous qualities:

- For instance, should access to customer database be isolated/cached?
- What happens if response does not arrive in time?
- How does the lifecycle management of instances work: create, delete, pool (per application, session, request)?

Questions to ask per arrow (in analysis, in design, in review), reasoning about load and miscellaneous qualities:

- Should access to customer database be isolated/cached?
- What happens if response from network does not arrive in time?
- How many data elements are transferred, and what are their properties (structure, protection needs, size)?

CID examples The arc42 documentation template has a Runtime View as its Section 6[6]. An example using PlantUML for low-level interaction visualizations is given here[7].

A CID from the "T" project is shown in the figure "CID (UML Sequence Diagram)".

[6] http://docs.arc42.org/section-6/
[7] https://dev.to/danlebrero/documenting-your-architecture-wireshark-plantuml-and-a-repl-to-glue-them-all

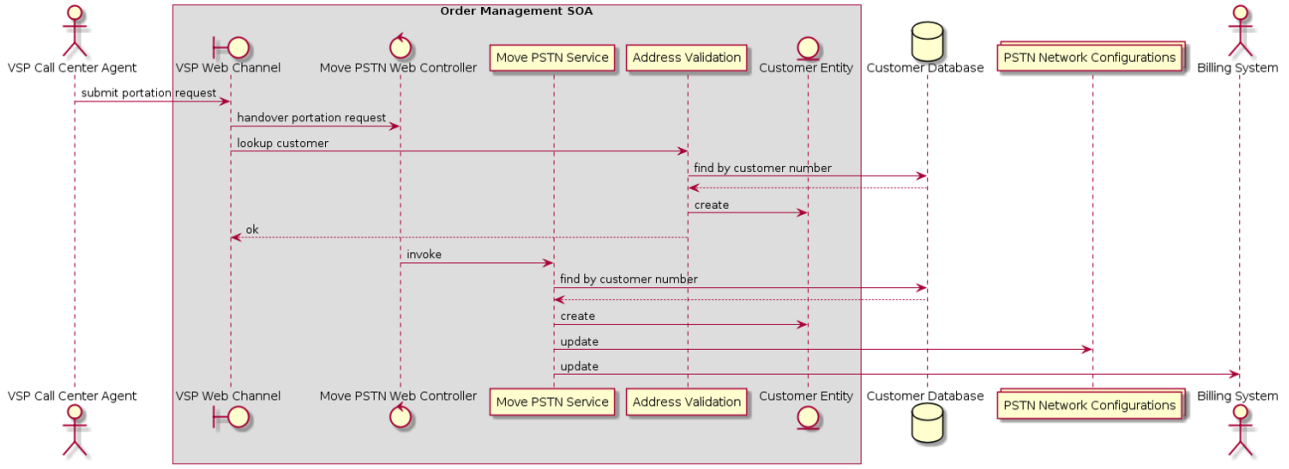

Figure 4: *CID (UML Sequence Diagram)*

Tips and Tricks

- Value consistency (no contradictions) over completeness (know it all)
- Good component descriptions should be SMART (like goals, but adapted)
- Each outgoing collaboration relationship must correspond to an incoming one elsewhere in system or its context (service consumer and provider)
- Sunny day and rainy days to be taken into account (remember the QASs?)
- Model on same level of detail on all cards ("if in doubt leave it out") and find a "medium ground": Overly precise specifications are hard to implement and to change. Vague ones do not add value, and their implementations are hard to integrate, the resulting architecture is difficult to evaluate.

- Never forget project vision or target audience: model with a purpose, and only create diagrams that serve an information need.
- What is in a name? Names should communicate what application/architecture are about. Metaphors are good, but must be chosen wisely (stakeholder reaction?). Strong semantics are preferred, e.g. "Web Browser" over "(Page) Client". One recommended naming schema is: domain concept plus architectural role/pattern.

More Information

Related Topics and Concepts

Miscellaneous Links:

- The Agile Alliance has a glossary entry for CRC cards[8].
- S. Ambler also features CRD cards in the context of agile modeling[9].
- RDD tutorial[10]

More links are available via the IFS website Method Selection and Tailoring Guide[11].

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-*

[8] https://www.agilealliance.org/glossary/crc-cards/
[9] http://agilemodeling.com/artifacts/crcModel.htm
[10] http://www.wirfs-brock.com/PDFs/A_Brief-Tour-of-RDD.pdf
[11] https://www.ifs.hsr.ch/index.php?id=13195&L=4

Oriented Software Architecture - Volume 1: A System of Patterns. Wiley Publishing.

Cheesman, John, and John Daniels. 2000. *UML Components: A Simple Process for Specifying Component-Based Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Starke, Gernot. 2015. *Effektive Software-Architekturen*. 7. Auflage. Hanser-Verlag.

Wirfs-Brock, Rebecca, and Alan McKean. 2002. *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education.